Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

Fertilicalc Mobile: A Flutter app for fertilizer management

Fertilicalc Mobile: Aplicación en Flutter para gestión de fertilizantes

Autor: Juan Villalobos Carrasco

Tutor: Juan Manuel Vozmediano Torres

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Fertilicalc Mobile: A Flutter app for fertilizer management

Autor:
Juan Villalobos Carrasco

Tutor:
Juan Manuel Vozmediano Torres
Profesor titular

Dpto. de Ingeniería de Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Grado: Fertilicalc Mobile: A Flutter app for fertilizer management

Autor: Juan Villalobos Carrasco

Tutor: Juan Manuel Vozmediano Torres

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

Acknowledgements

First and foremost, I would like to thank my dad, whose encouragement and support were invaluable.

My deepest appreciation goes to the friends who have accompanied me along the way: Víctor, Aurora, Luis, Pedro, Fran, Esther, César, Manu… They helped keep me determined throughout the long evenings spent on this project.

Many thanks to my tutor for his valuable advice, and to the hard-working people at ETSI (and FCOM) for assisting me in the wonderful journey of becoming a graduate engineer.

I also owe a great debt of gratitude to the amazing contributors to the open-source Flutter community, and to Stack Overflow.

Juan Villalobos Carrasco

Sevilla, 2021

Abstract

In this project, we design and develop a mobile app that adapts the desktop software *Fertilicalc*. This app, which we call *Fertilicalc Mobile*, allows farmers and agriculture professionals and students to calculate the nutrient and fertilizer requirements for a chosen selection of crops.

The app is built on Flutter, a multiplatform framework that allows us to produce Android and iOS versions of the app, with native performance. We propose a '*Model – View – Commands + Services*' architecture, based on state management tool *Provider*, around which our app is built.

To implement the functionality of *Fertilicalc*, we set up a REST API, which executes a Python script that performs the necessary calculations for our app.

The result of the project is a functional and extensible fertilizer management app, which also has an educational component.

Resumen

En este proyecto diseñamos y desarrollamos una aplicación móvil que adapta el software de escritorio *Fertilicalc*. Esta aplicación, que llamamos *Fertilicalc Mobile*, permite a agricultores y estudiantes de agricultura calcular las necesidades de nutrientes y fertilizantes para una selección de cultivos.

La aplicación está construida sobre Flutter, un *framework* multiplataforma que permite producir versiones de la aplicación en iOS y Android, con rendimiento nativo. Proponemos una arquitectura '*Modelo – Vista – Comandos + Servicios*', basada en la herramienta de gestión de estado *Provider*, alrededor de la cual hemos construido nuestra aplicación.

Para implementar la funcionalidad de *Fertilicalc*, hemos instalado una API REST que ejecuta un script de Python, el cual realiza los cálculos necesarios para nuestra aplicación.

El resultado de este proyecto es una aplicación funcional y extensible para la gestión de fertilizantes, que cuenta además con un componente educativo.

Index

LIST OF FIGURES

LIST OF CODE SNIPPETS

1 INTRODUCTION

Did you hear about the rose that grew
from a crack in the concrete?
Proving nature's law is wrong it
learned to walk without having feet.
Funny it seems, but by keeping its dreams,
it learned to breathe fresh air.
Long live the rose that grew from concrete
when no one else ever cared.
- Tupac Shakur -

Global food security is one of the most pressing challenges facing humanity, and commercial fertilizers will play a critical role in achieving it. [1] The world population is projected to reach 9.7 billion people by 2050, and agricultural production will need to almost double at that time. [2]

In developing countries, especially, fertilizer availability and affordability are critical to increasing agriculture production. By increasing yield per unit area, the use of fertilizers will spare millions of acres of ecologically sensitive land that otherwise would have to be converted to agriculture. [1]

It is crucial, however, that this intensified production is achieved in the most sustainable manner, without sacrificing the ecological integrity of the food system. To this end, the development of fertilizer best management practices is paramount. One such example would be the implementation of the International Plant Nutrition Institute-initiated "4Rs". This is a management practice for farmers, which includes: right source/kind (match the fertilizer type to crop needs); right time (make nutrients available when the crop needs them); right rate (match the amount of fertilizer to crop needs); and right placement (keep nutrients where crops can use them). [1]

Rational fertilization management is critical for efficient crop production and the efficient use of non-renewable resources, which can have a great environmental impact. However, there are not many simple decision-making tools in the market for this purpose. [3]

Fertilicalc

Fertilicalc is a Windows program for calculating nutrient and fertilizer requirements of crops. It allows the user to calculate the amounts of nitrogen (N), potassium (P), and phosphorus (K) needed and the most cost-effective combination of commercial fertilizers for up to 149 crops. [3][4]

Its objective is to be a useful tool for farmers and agronomists in estimating nutrient requirements for different

crops and in the selection of fertilizers. It also has an educational aim, helping students to better understand the rationale behind fertilization management. [3] [5]

The program allows the user to select different fertilizers to determine the necessary rates to apply to the chosen crops. The user must supply a yield for each crop he chooses, select some fertilizers to apply to it, and specify the values of certain parameters, which are used to adjust the calculation. [5]

As part of its output, *Fertilicalc* provides to the user the rates of specific fertilizer products to apply to a given crop; this covers the "right rate" practice outlined previously. [5]

Fertilicalc Mobile

There are currently an estimated 5.27 billion unique mobile phone users worldwide, growing at a rate of 1.9 percent per year. [6] In many emerging and developing countries, most people's only Internet access is through a mobile device; in some cases, they have skipped landlines altogether and moved straight to mobile technology. [7]

Therefore, we have considered it particularly valuable to adapt *Fertilicalc* to a mobile platform; this would allow it to reach a vast number of users that it otherwise could not. Furthermore, users in these developing countries, where the livelihood of a large percentage of the population depends on farming [8], are the ones who could benefit the most from a simple fertilizer management tool such as this one.

This project aims to adapt *Fertilicalc* into a mobile app that is fast, extensible, and easy to use. To integrate the functionality of the desktop app, we will use a Python script that performs all the necessary calculations based on the user's input. This script was provided courtesy of the author of *Fertilicalc*; a version of it is publicly available at the *Fertilicalc* website [4]

To use the *Fertilicalc* script as an external resource, we have modified it so that it takes a JSON input and returns a JSON output. We have deployed a REST API server, using micro web framework Flask, that runs the script for the app and returns the results in the HTTP response (this mechanism is detailed in Appendix A).

Additionally, the app consults botanical API Flora Codex to retrieve some additional data on crops, which is shown to the user for educational purposes. [36]

The diagram below illustrates the interaction between the app and these two APIs[1]:

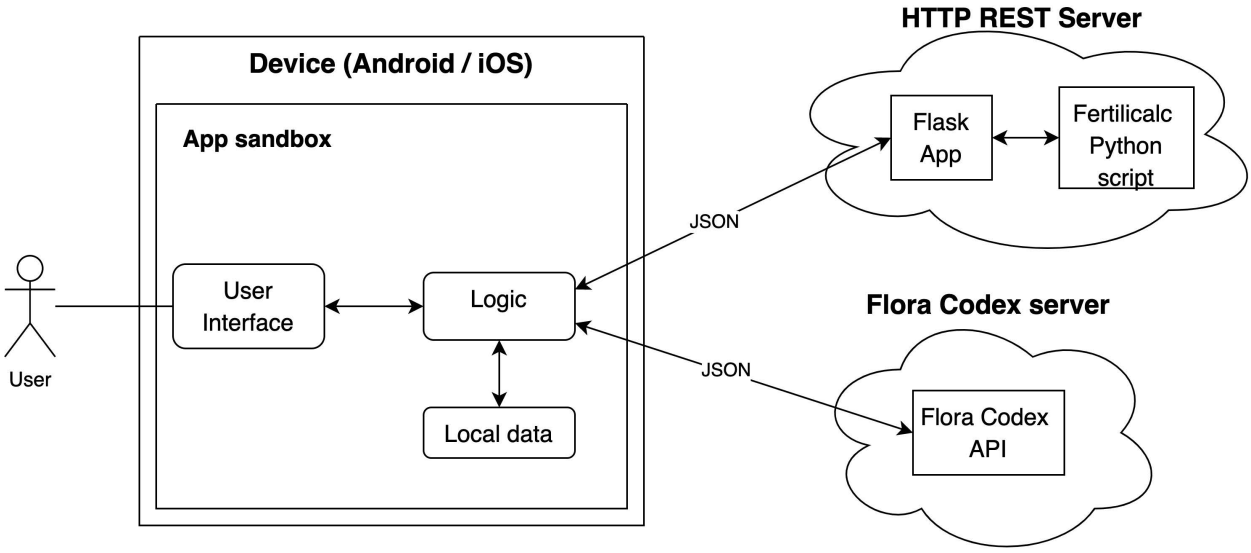


Figure 1-1. Simplified systems diagram

[1] The internal architecture of the app has been simplified; the actual architecture of the app is exemplified in figure 5-14.

For building the app, we have chosen the cross-platform framework Flutter. Producing a native app for both Android and iOS would require the development of two apps and support for two codebases. Meanwhile, cross-platform development with Flutter provides support for both Android and iOS with native performance, allowing for more cost-effective and quick development. Essentially, it allows us to maximize the user reach while also maximizing the development speed.

The versatility of the Flutter framework and its vast array of third-party packages also allow us to implement features that would be otherwise too complex to include. For instance, by implementing one of these packages, we have included a screen where the user can choose his location through a Google Maps-like interface (see section 5.2.5).

Project development

This project started in July of 2020 with an estimated duration of 1 year. Below is an approximate account of the work schedule that was followed:

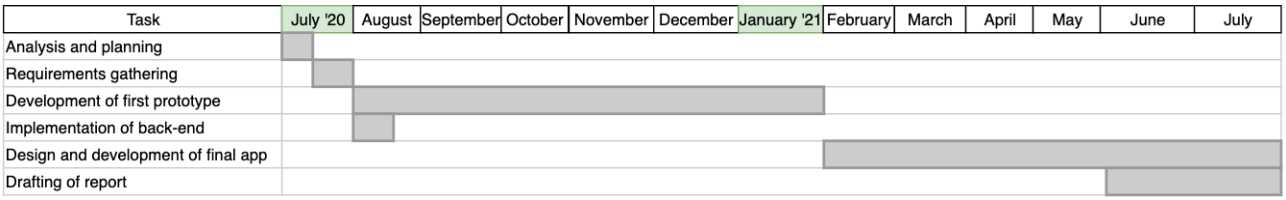


Figure 1-2. Work schedule

- **Analysis and planning** (2 weeks): Initial research on Flutter and Fertilicalc, and project planning.
- **Requirements gathering** (2 weeks): Establishment of the app's objectives, drafting of requirements, and sketching of screen mockups.
- **Development of prototype** (6 months): Development of an initial prototype, early-stage design of the app, and research on the Flutter framework.
- **Implementation of back-end** (2 weeks): Installation of REST API to deploy the Fertilicalc Python script that our app interacts with to calculate results.
- **Design and development of final app** (5 months): Adoption of a new architecture (described in chapter 4), design of the *pre-release* version of the app, and incorporation of additional features.
- **Drafting of the report** (2 months).

Resources

This project was made on an early-2015 MacBook Air running macOS Catalina. The following were the mobile devices used for testing throughout the development of the app:

- OnePlus 6 running Android 11.
- Samsung Galaxy S9 running Android 10.
- Samsung Galaxy S6 Edge running Android 7.

For testing on iOS, the XCode simulator was used to run the app on a virtual iPhone 12 Pro Max device. Below are screenshots of the app running on different devices, from left to right: Samsung Galaxy S6 Edge, OnePlus 6, and iPhone 12 Pro Max (virtual).

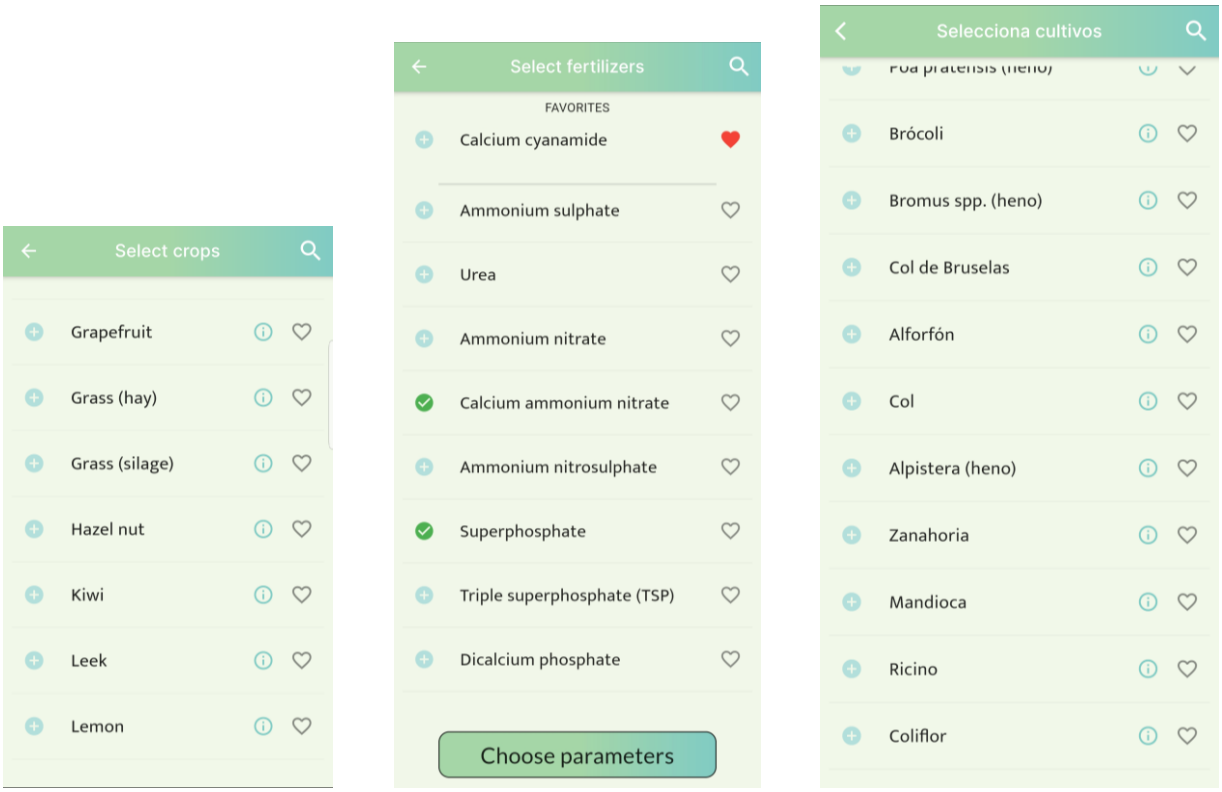


Figure 1-3. App running on different devices

The IDE used for development was Visual Studio Code, and Git was used for version control. Below is a screenshot of the workspace where the app was developed; the '*timeline*' on the bottom left shows a list of some of the project's Git commits.

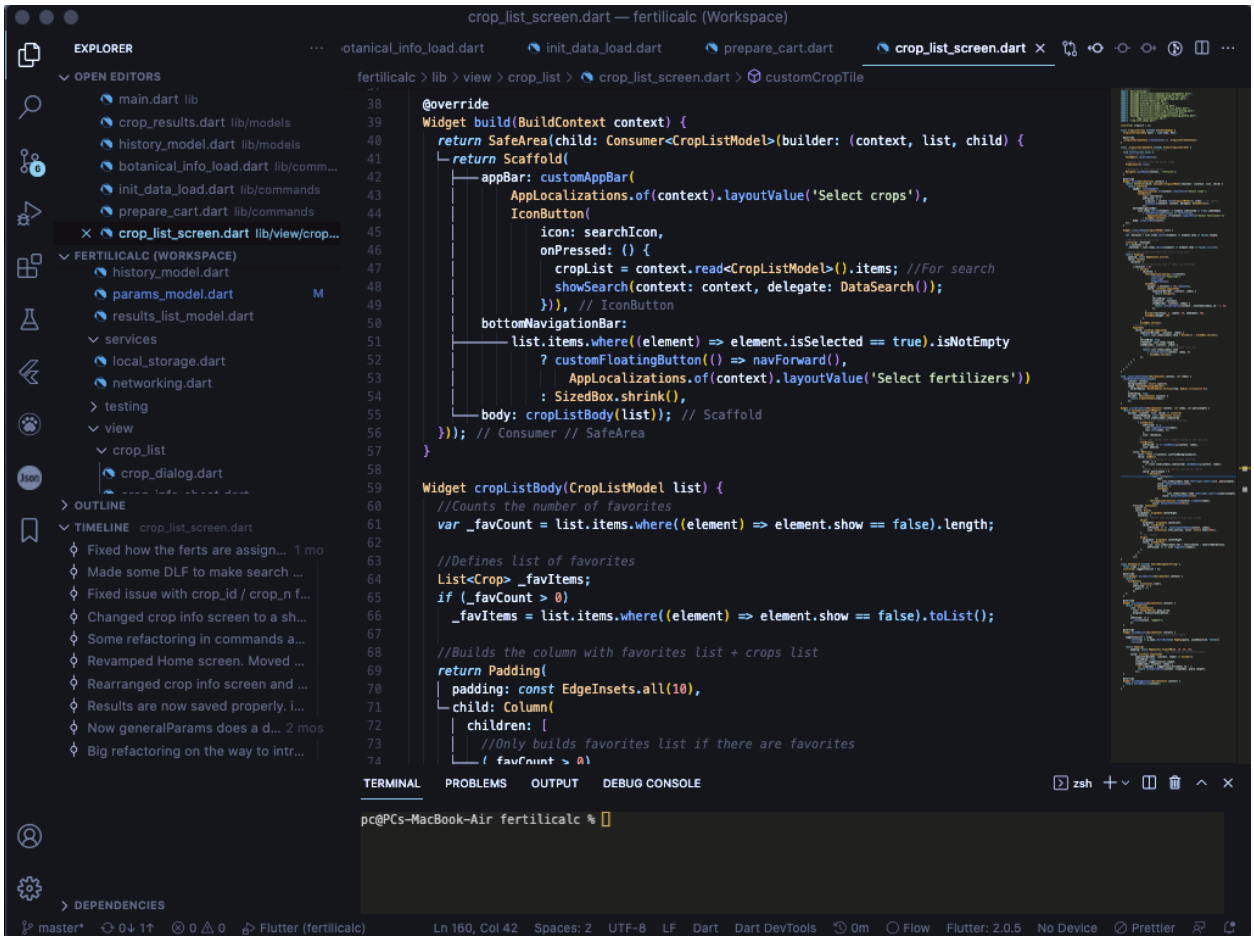


Figure 1-4. Visual Studio Code workspace

For drawing most of the diagrams that appear throughout this document, online tool *diagrams.net* (also known as *draw.io*) was used [9]. For setting up the web server that hosts the REST API which executes the *Fertilicalc* script, we used the online service *pythonanywhere.com* [10].

Report structure

The following is a summary of the different sections that this report comprises:

1. **Introduction.** First approximation of the problem, its context, and the nature of the developed solution.
2. **Development framework.** Advantages of cross-platform development, comparison of Flutter and React Native, and basic concepts of the Flutter framework, useful for understanding the architecture and design of the app.
3. **Requirements.** Description of the objectives and functionality of the app.
4. **Architecture.** Introduction of app state management with Provider, and description of the architectural pattern around which the app is designed.
5. **Design and Development.** Description of the developed solution, delineating its different modules and the implementation of the requirements.
6. **Conclusions and Future Work.** Reflection on the accomplishment of the project's objectives, and proposal of future improvements.

2 DEVELOPMENT FRAMEWORK

Cross-platform development refers to the process of creating an app that works on several platforms. This type of development is especially attractive for small teams since it offers an inherent reduction in development time and costs. [11]

Native app development, in contrast, involves building an app exclusively for a single platform. In the case of mobile apps, there are two main platforms: Android, which uses Java or Kotlin, and iOS, which uses Objective-C or Swift.

In this chapter, we motivate our decision to choose cross-platform development, and specifically Flutter, as our framework. Then, we introduce basic concepts regarding Flutter that serve as a basis for understanding the architecture, design, and implementation of our project.

2.1 Cross-platform framework

In most cases, developers will want to reach the broadest possible audience with their app. To ensure this, the application should be released on multiple platforms. If the development team chooses native development, they will need to maintain multiple codebases; this leads to more maintenance, more effort, and ultimately more cost.

Cross-platform is more cost-effective than native since it allows us to deploy in different platforms using a single SDK (Software Development Kit) tool while achieving similar performance to a native app. Only a single code base is created, which targets both platforms, so development time is faster and code reusability is improved (iOS code cannot be reused by the Android app and vice-versa).

Another advantage of choosing cross-platform is that the user experience, while it may not be as refined as with native, should be more uniform and better aligned between platforms. When features are added or changes are made, they will generally manifest on both versions at the same time.

Most importantly, choosing cross-platform allows us to broaden the market reach far more quickly. The global market share of Android and iOS is 72.84% and 26.34%, respectively (as of June 2021). [12] Even though our app is especially oriented toward users in developing countries, where the Android market share is even greater [13], the production of an iOS version reinforces our objective of reaching the largest user base possible.

The two most widely used cross-platform mobile frameworks are Flutter and React Native. According to a 2021 global developer survey, Flutter is used by 42 percent of cross-platform developers, while React Native is used by 38 percent. [14]

React Native was released in March 2015, while Flutter was released in May 2017. Consequently, the React Native community has produced a larger number of ready-to-use libraries and packages. Flutter, on the other hand, has been comprehensively supported by the Google team from its beginning and has excellent documentation and a vibrant developer community. [15]

Flutter has shown substantial growth since its release, with its adoption growing quicker than React Native. As of July 2021, Flutter has approximately 30% more *GitHub Stars*, a metric that illustrates the popularity of open-source projects among developers. The rapidly growing interest in Flutter is also shown when comparing the trends in Google Search for the last 4 years, as seen in the figure below.

Figure 2-1. Google Search trends of Flutter and React Native during last 4 years [16]

Apps made with React Native include Instagram, Facebook, and Tesla; while currently, popular Flutter apps include Google Ads, Xianyu by Alibaba, and Reflectly.

It is commonly believed that cross-platform apps provide poorer performance than native. However, the difference may be considered fairly insignificant, especially for small and medium-sized apps. In tests made on physical devices, Flutter demonstrated just 9-22% lower performance in memory-intensive tests in comparison to native. React Native, on the other hand, showed the worst performance in all the tests, being 2–15 times slower than Flutter and 5–21 times slower than Native. [17]

This can be partly explained by the architectural principles of both frameworks. The execution of code on React Native requires a "bridge" from the JavaScript code to the device's native environment. This bridge translates JS code to the device's native programming language and vice-versa, requiring extra time and resources for processing. In contrast to this, Flutter compiles to native libraries without any additional intermediate layers. Therefore, it performs quicker and consumes fewer resources to execute the code. [15]

In sum, we have chosen Flutter over React Native due to its rising popularity, its smooth performance, and its extensive documentation and support. Along the development of the project, we have demonstrated additional benefits, such as the simplicity and modularity of the *widget* ecosystem (explained later in this chapter) and the extensive array of third-party packages built by the Flutter community. The latter has allowed us to build some parts of the app especially quickly and implement features that would otherwise be too labor-intensive.

In the next section, we provide additional details and basic concepts related to the Flutter framework.

2.2 Basics of Flutter

Flutter is an open-source UI software development kit made by Google. It is used to build natively compiled applications for mobile (both iOS and Android), web, and desktop from a single codebase. Flutter provides a

simple way of composing User Interfaces, focusing on appearance, behavior, and integration between both.[2]

Flutter apps are written in the Dart language. Also developed by Google, it is an object-oriented language with C-style syntax that focuses on front-end development. While writing and debugging an app, Flutter uses Just-In-Time compilation, allowing for "Stateful Hot Reload". This means that, when updating source code files, the Flutter framework automatically rebuilds the interface, allowing the developer to immediately view the effects of the changes. Therefore, the developer can quickly and easily experiment, build UIs, add features, and fix bugs.

The Flutter engine is written primarily in C++ and provides rendering support using Google's Skia graphics library and interfaces with platform-specific SDKs such as those provided by Android and iOS. The Flutter engine is a portable runtime for hosting Flutter applications; it implements Flutter's core libraries and contains everything needed for app development.

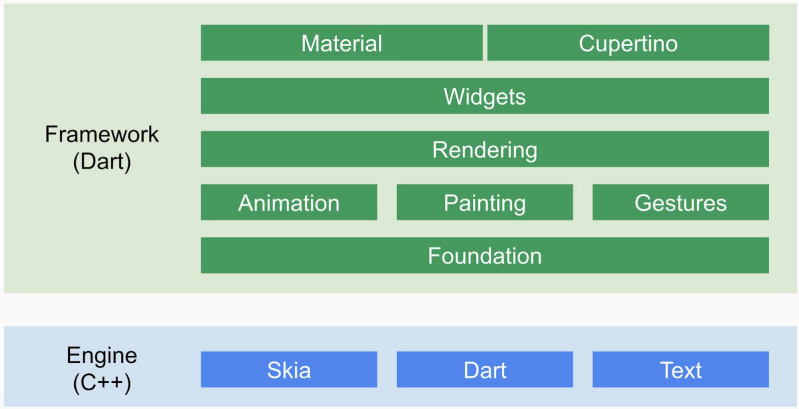


Figure 2-2. High-level overview of the Flutter framework and engine [18]

One of Flutter's main differences from other frameworks such as Android is that its UI is built in code, not in an XML file or similar. Flutter is a "*declarative*" framework.

In an "*imperative*" framework (e.g., Android, iOS, or web), the user interface is explicitly called in code. In a "*declarative*" framework such as Flutter, the code declares that the user interface should look a certain way, given a certain state. The user interface of a Flutter app can be seen as the result of one function that takes in the current state of the app as a parameter.

The central idea of the architecture is that the UI is built of widgets. Almost everything in Flutter is a widget; they're used for both UI elements (e.g., `Text` or `Image`) and layout (e.g., `Row` or `Column`).

A widget describes what its view should look like with its current configuration and state. When its state changes, the widget rebuilds its description, and the framework compares it against the previous description to perform the minimal changes needed in the underlying render to transition from one state to the next. A widget's main job is to implement a `build()` method which describes the widget in terms of other, lower-level widgets (its descendants), which are built by the framework in order to render the UI on the screen.

When writing an app, it is common to create new widgets that are subclasses of either `StatelessWidget` or `StatefulWidget`, depending on whether the widget manages any state.

A **Stateless Widget** is a simple UI component that displays only the data it is given; it has no "memory". A **Stateful Widget**, on the other hand, is one that the user can interact with, and which has a memory. Its state is stored in a State object; when it changes, the State object calls `setState()`, telling the framework to redraw the widget. A major advantage of this is that one can just rebuild anything that needs updating rather than having to individually change instances of widgets.

In our app, each screen is built as a Stateful Widget. Even the app itself is a Stateful Widget, which allows us to

[2] Most of the introduction to the Flutter framework included in this section is based on the official documentation from the Flutter website [45]

rebuild all screens when the language or another global setting is changed.

The Flutter framework provides a range of powerful basic widgets; some of the most used are `Text`, `Row`, `Column`, `Container`, etc. Additionally, widgets can be incorporated from external third-party packages.

A Flutter app is essentially a tree of nested widgets, such as the one illustrated in the following figure:

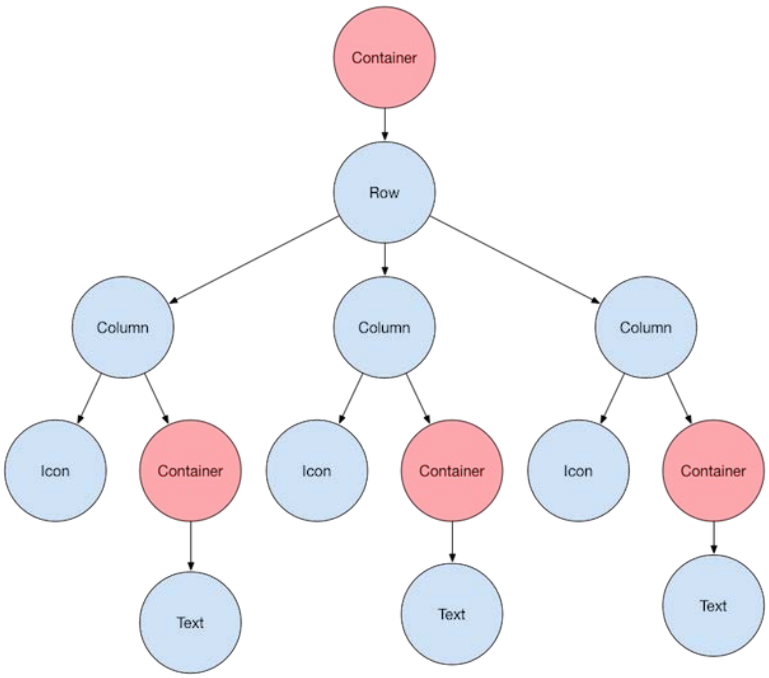


Figure 2-3. Diagram of a widget tree [19]

The screen's layout is created by composing widgets to build more complex widgets. Layout widgets, such as `Row` or `Column`, are in charge of arranging, constraining, and aligning the visible widgets they contain. The `Container` widget, also widely used, creates a rectangular visual element, applying margins, padding, and constraints to its child (e.g., a `Text` widget).

Flutter's widgets incorporate critical platform differences such as scrolling, navigation, icons, and fonts to provide full native performance on both iOS and Android.

Below is an example of a minimal app on Flutter:

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Welcome to Flutter'),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

Minimal Flutter app example [20]

The app class (`MyApp`) extends `StatelessWidget`, which makes the app itself a widget. Material Design, which the `MaterialApp` widget and its descendants implement, is a visual design language that is standard on mobile and the web. Flutter offers a rich set of *Material* widgets, appropriate for Android's style, as well as *Cupertino* widgets, suited for iOS style.

The `Scaffold` widget, from the `material` library (one of the framework's foundational libraries), provides a default app bar, and its `body` property holds the widget tree for the home screen. In this case, the body encapsulates a `Center` widget, which aligns its child to the center of the screen. Its child, a `Text` widget, shows the corresponding text on the screen.

Below is the widget tree for this screen and its appearance on Android and iOS devices:

Figure 2-4. Widget tree and appearance of the minimal app (screenshots from [20])

The widget tree can quickly become quite complex; for this reason, in our project, we will divide some of the more complex screens into multiple widgets. For instance, to build our app's Crop List Screen, we define the screen's body in terms of smaller custom widgets (see section 5.2.2).

We will also take advantage of the modularity of the widget ecosystem to improve code reusability and cleanliness. For this, we will define some of our custom widgets so that we can reuse them on different screens. One example is our `CustomAppBar` widget:

```
import 'package:flutter/material.dart';

AppBar CustomAppBar(String title, Widget actions) {
  return AppBar(
      flexibleSpace: Container(
```

```
        decoration: BoxDecoration(
          gradient: LinearGradient(
              colors: [appBarColor1, appBarColor2], stops: [0.5, 1.0]),
        ),
      ),
      title: Text(title,
          style: TextStyle(color: Colors.white)),
      centerTitle: true,
      actions: [actions]);
}
```

CustomAppBar widget

The `appBar` property of a `Scaffold` widget takes an `AppBar` widget as an argument. Our custom `AppBar` is built according to two parameters: a title, and actions (widgets that appear on the right, e.g., a search button). The `AppBar` widget itself is provided by the `material` library. Properties that we do not customize are set by default; for instance, if the `leading` property (widget that appears on the left) is omitted and the screen has any previous routes, a `BackButton` is inserted, which can be used to navigate to the previous screen. [21]

Below are examples of two screens in our app where `CustomAppBar` is used, along with the code used in each case to instantiate it:

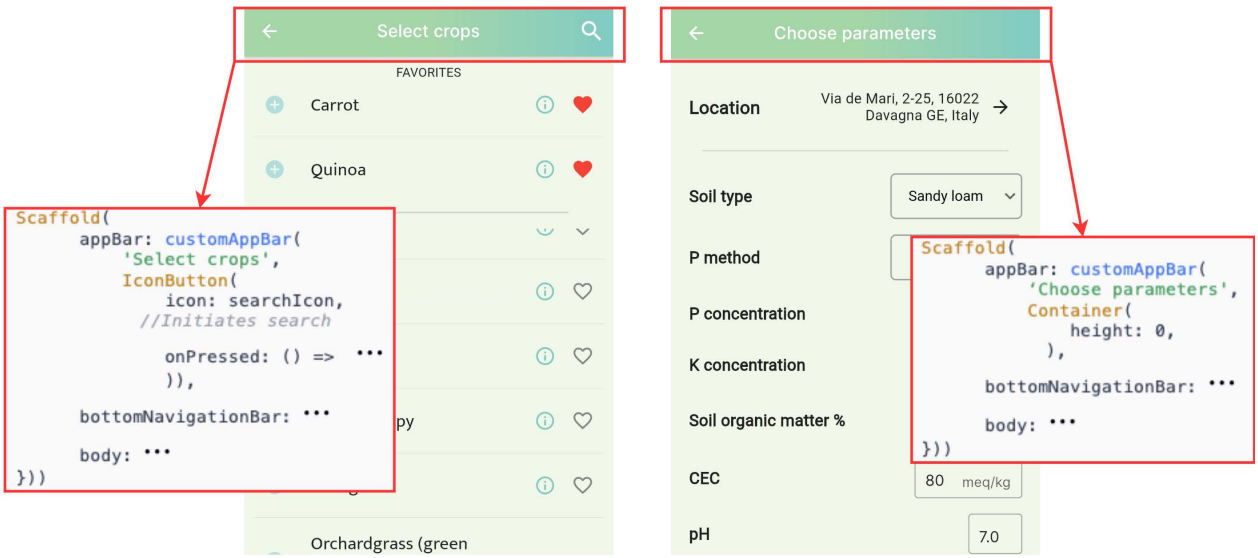


Figure 2-5. Uses of CustomAppBar

Another custom widget we have defined and reused in different screens is `CustomFloatingButton`:

```
import 'package:flutter/material.dart';

Widget CustomFloatingButton(VoidCallback navCallback, String title) {
  return Padding(
    padding: const EdgeInsets.fromLTRB(50, 12, 50, 12),
    child: InkWell(
      onTap: navCallback,
      child: Container(
        child: Text(title,
            textAlign: TextAlign.center, style: bottomAppBarText(25)),
        decoration: BoxDecoration(
          border: Border.all(width: 1.0),
          borderRadius: BorderRadius.circular(12),
          gradient: LinearGradient(
              colors: [appBarColor1, appBarColor2], stops: [0.5, 1.0]),
  ),   ),  ),  );  }
```

CustomFloatingButton widget

This widget builds a button which we use on different screens to navigate to the next screen. Since we reuse it, it is built uniformly across screens, only differing in two parameters: a title, and a callback which specifies the function to be called when the button is pressed (`onTap` property).

Below are examples of screens where our `CustomFloatingButton` is used:

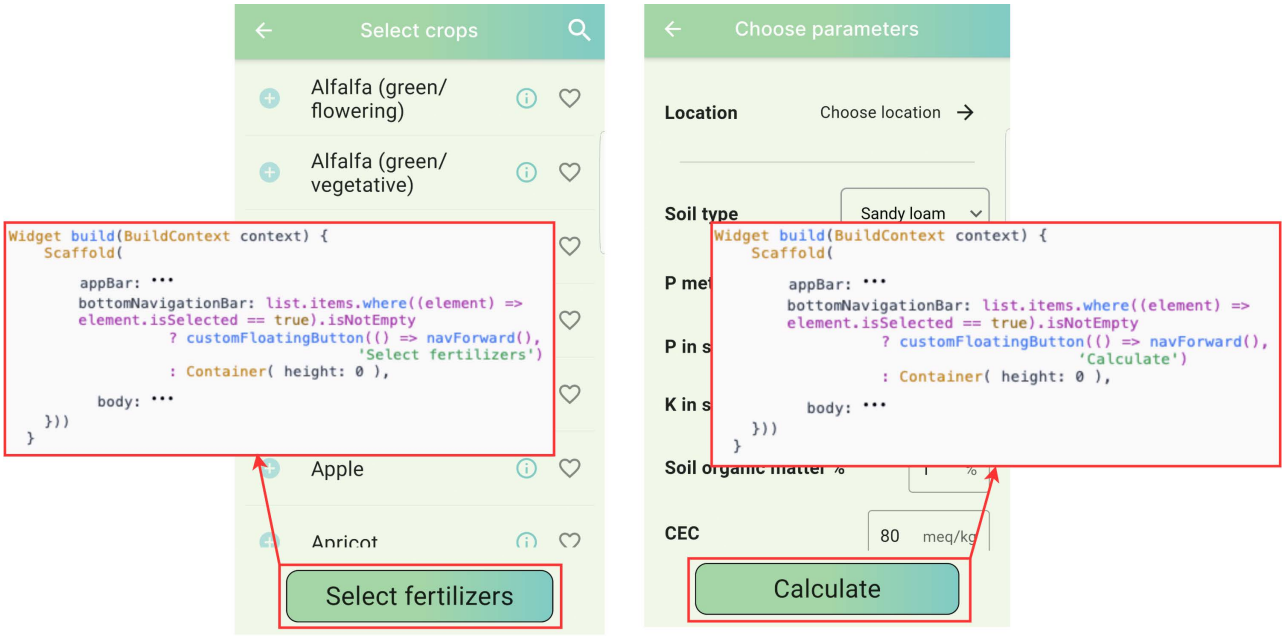


Figure 2-6. Uses of CustomFloatingButton

In both cases, the callback is defined as a call to a `navForward()` function, which is defined independently by each screen to perform the appropriate navigation.

The following are some other basic concepts regarding Flutter, which we approach later in this document:

- App state management: In particular, we focus on the solutions offered by the *Provider* package (section 4.2 and throughout chapter 5)
- Navigation: Use of the framework's *Navigator* tool to navigate to different screens (section 5.2.2)
- Construction of a list with the `ListView` widget (section 5.2.2)
- Construction of a dialog and a bottom sheet (section 5.2.2)
- JSON serialization (section 5.3)
- Data storage (section 5.4.1)
- HTTP networking (section 5.4.2)
- Internationalization using the `AppLocalizations` tool (section 5.6)
- Use of various third-party packages, e.g., *Permission Handler* (section 5.2.4), *Google Maps Place Picker* (section 5.2.5), *Shared Preferences* (section 5.4.1), etc.

3 REQUIREMENTS

The purpose of requirements documentation is to provide a detailed description of the requirements needed to complete the product (our app), which helps communicate and define its scope. It states the product's purpose and what it must achieve, but not how to deliver or build what is needed.

In our project, it was an iterative process that evolved in parallel with design and implementation. Some of the functional requirements of the project were added after development was underway, and more will be added as the app continues to be developed. Detailed here are the ones that were included in the final product.

Note that the essence of the app and, subsequently, most of the requirements, are the result of adapting the functionality of the Fertilicalc desktop app to the mobile framework.

3.1 General description

Basic app description

The app allows the user to select some crops, along with some fertilizer products to apply to these, and customize some parameters, like his location or soil data. From all this input, the user will obtain the rates of fertilizer to apply to each crop, and some additional data useful for fertilization management.

User characteristics

The app is mainly intended to be used by farmers, especially in developing countries. The app is also targeted towards agriculture professionals (in the academic or business fields) and agriculture students around the world.

A typical user should have some basic experience using mobile applications and mobile operating systems, and intermediate knowledge of agriculture terminology, to use the application effectively.

User objectives

The application allows the user to make a calculation for a rotation[3] of certain crops with certain characteristics. For this calculation, the user adds crops, along with their yield, then adds fertilizers for each of these crops. Additionally, the user can customize the crop rotation with data such as the location, the type of soil, or the water supply.

When the calculation has been performed, the app displays the rate of each chosen fertilizer to be applied to each

[3] A '*crop rotation*' is the practice of growing a series of different crops in the same land across successive seasons or years. For our app's purposes, a rotation simply involves a list of different crops.

chosen crop, along with other information useful for fertilization management, such as acidification or the nutrient requirements.

The user can save these results and retrieve them later, as well as share them outside the application.

3.2 Functional requirements

Criticality Scale

Less Critical	Less
Critical	Normal
Very Critical	Very

ID	Description	Criticality	Dependencies
R-01	On one of its screens, the app displays a list of available crops, each identified by a name and accompanied by at least one button, which is tapped to select the crop and potentially add it to the rotation.	Very	
R-02	When selecting a crop, the user can add it to the rotation by entering a positive yield.	Very	R-01
R-03	When selecting a crop, the user can customize the following parameters[4]: • Percentage of residues left in the field (integer) • Burning of residues (Boolean)	Normal	R-01
R-04	The app may display a search bar through which the user can search for a specific crop from the list of available crops and select it.	Normal	R-01
R-05	The user can mark any crop as a favorite, and later unmark it. The associated favorites selection persists between executions.	Less	R-01
R-06	For each crop in the crop list, the app can display, at the user's request, the following information about the crop: species, genus, and family. It is also able to display a photo of the crop.	Less	R-01
R-07	In one of its screens, the app displays a list of available fertilizer products, each identified by a name and accompanied by at least one button, which is tapped to select the fertilizer and potentially add it to one or more of the crops in the rotation.	Very	

[4] A more detailed description of inputs and outputs of the app can be found at [4]

R-08	The user can begin to select fertilizers when he has added at least one crop to the rotation.	Very	R-01, R-07
R-09	When selecting a fertilizer, the user can customize the following parameters: • Incorporated (Boolean) • Before planting (Boolean)	Normal	R-07
R-10	When selecting a fertilizer, the user can select which crops (from those previously selected) to apply it to.	Normal	R-01, R-07
R-11	The app may display a search bar through which the user can search for a specific fertilizer from the list of available fertilizers and select it.	Normal	R-07
R-12	The user can mark any fertilizer as a favorite, and later unmark it. The associated favorites selection persists between executions.	Less	R-07
R-13	The user can choose the location for the calculation from a map.	Very	
R-14	Once the user chooses a location, it is saved as default and persists between executions.	Normal	R-13
R-15	The user can customize the following global parameters for the calculation: • Soil type (4 possible values) • P method (7 possible values) • P concentration (integer) • K concentration (integer) • Percentage of soil organic matter (integer) • CEC (integer) • pH (integer) • Water supply (2 possible values) • Strategy (4 possible values can take) • Tillage (Boolean)	Very	
R-16	The app initially displays default values for the global parameters. When the user changes these, the new values are saved and persist between executions.	Normal	R-15
R-17	Default values of global parameters can be restored.	Normal	R-15, R-16
R-18	The user can trigger the calculation of results by tapping a button, only when at least one crop and one fertilizer have been added to the rotation, and a location has been chosen.	Very	R-01, R-07, R-13
R-19	Once results have been calculated for a rotation, the app displays these results.	Very	R-18

R-20	The results displayed for a calculation include, for each crop in the rotation: • The rate of each fertilizer chosen for that crop • Acidification • N loss • N average • K oxide • P oxide	Very	R-19
R-21	The user can save the results of a calculation for later retrieval.	Normal	R-19
R-22	On one of its screens, the app presents to the user a history of previously saved results.	Normal	R-21
R-23	The user can consult any results from history, accessing all the information that was shown when results were calculated and displayed for the first time.	Normal	R-21, R-22
R-24	The user can share results (either newly calculated or from history) with a third party.	Less	R-19
R-25	The user can delete specific results from history.	Less	R-21
R-26	The user can delete the entire history of results.	Normal	R-21
R-27	The user can delete all data stored internally, e.g., his location, favorites, etc.	Normal	
R-28	On one of its screens, the app allows the user to change the language of the app.	Very	

Figure 3-1. Table of functional requirements

3.3 Non-functional requirements

- **Performance.** The application should be less than 20 Mb in size.
- **Usability.** The difficulty of the user to understand the app should be easy to intermediate. To aid in this aspect, an in-app tutorial should be included, in image or video form.
- **Internationalization.** The app will be available in 7 languages: English, Spanish, Italian, Russian, Hindi, Arabic, and Chinese.

  When set to a particular language, all texts throughout the app will be shown in the chosen language, except for select technical terms. When running in Arab, the User Interface will have a Right-To-Left presentation.
- **Compatibility.** The app should run on Android versions newer than Android 5.0 and iOS versions newer than 12.4. The user interface should be adequately displayed for devices with screens as small as 5.1 inches (*Samsung Galaxy S6 Edge*), and as large as 6.68 inches (*iPhone 12 Pro Max*).

4 ARCHITECTURE

Mobile app architecture can be defined as the set of techniques and patterns that are supposed to be followed for building a structured mobile app ecosystem. It defines an app's skeleton, upon which the different structural elements and their relationship with the framework are based. [22]

In this chapter, we describe our choice of state management approach, which is a central aspect of the app's architecture. Then, we propose an architectural pattern around which the app will be built.

4.1 App state management

A critical concept in mobile app architecture is app state, i.e., state that is shared across many parts of the app, and that is kept between user sessions. In Flutter, state management refers to the way an app handles the distribution of state through the widget tree. It deals with how different parts of the app interact with one another and how mutable data is stored and accessed. [23]

Choosing a state management solution is central to building an app. It should be one of the earliest decisions made, as it is nontrivial to retroactively change the solution later in the development process. Flutter does not impose any kind of architecture or state management solution.; this has led to the creation of multiple state-management solutions, and a handful of architectural approaches have originated from the community. [24]

The following is a comparison of the three most popular state management approaches:

Solution	***Short description***	***Pros***	***Cons***
Provider	Package that provides an improved interface for Flutter's inbuilt *Inherited Widget*. Gives the ability to provide state from a widget to all its descendants in the widget tree.	Good for small applications (little boilerplate code); easy to learn; endorsed multiple times by the Flutter team.	Has no related architecture; low scalability.
Redux	Architectural pattern that uses a "store" as the central location for all business logic. The UI should only send actions to the store (such as user inputs) and display the UI depending on the current state of the store.	Clearly defined rules; state changes are perfectly predictable; can be used to implement a three-layered architecture: *UI – Store – Data*	The store will get very large with large applications; has a high learning curve.

BLoC	Architectural pattern where all business logic is extracted from the UI into BLoCs (Business Logic Components). The UI should only publish events to the BLoCs and display the UI based on the state of the BLoCs.	Has clear architectural rules; state changes are perfectly predictable; endorsed multiple times by the Flutter team.	Has a high learning curve.

Figure 4-1. Comparison of state management solutions [24]

We decided to choose Provider for this project due to its simplicity, low learning curve, extensive documentation, and support by the Flutter team. Since our app is not particularly large (does not include many screens or a large set of features), we considered this was sufficient and preferable to a more sophisticated approach.

4.2 Provider

In Flutter, to make state accessible to multiple widgets, we "lift the state up" to an ancestor of all the widgets that are using that common state. This way, the state will be accessible to all its descendants (i.e., any widgets below them). [23]

Flutter has mechanisms for widgets to provide data and services to its descendants. Provider uses these low-level widgets but is simpler to implement. The following are three essential components of Provider:

- ***ChangeNotifier*:** Class that provides change notification to its listeners. In Provider, it is one way to encapsulate the application state, through '**models**' (i.e., objects that encapsulate data). There can be several *ChangeNotifier*, each one for a specific model. Below is an excerpt of `CropListModel`, which is one of the models used by our app:

```dart
import 'package:flutter/foundation.dart';
import 'package:fertilicalc/models/crop.dart';

class CropListModel extends ChangeNotifier {
  List<Crop> items;
  CropListModel({this.items});

  //Selects a crop from the list or unselects it
  void toggleSelected(int index) {
    items[index].isSelected = !items[index].isSelected;
    notifyListeners();
  }

  //Marks a crop as fav or unmarks it
  void toggleFav(int index) {
    items[index].fav = !items[index].fav;
    notifyListeners();
  }
    ...
```

CropListModel class[5]

This model stores the list of crops that the app will display and defines methods that are called when the user selects a specific crop or marks it as a favorite. The only code specific to *ChangeNotifier* is the `notifyListeners()` method, which is called any time the model changes in a way that might change the app's UI. In this case, the action of selecting or unselecting a crop from the list (by calling the `toggleSelected` method) will be reflected in the UI.

[5] For the sake of brevity, the Dart code shown throughout this document omits some lines (indicated with the ••• symbol). The full code of the project can be browsed at the private repository found at [45]

The figure below shows a portion of the screen that displays the list of crops, at different points in the execution. Initially, every crop is unselected; when the user taps the 'Add' button on one of them, the `toggleSelected` method is called, which in turn calls `notifyListeners`. This triggers a UI rebuild of any widget that was listening; in this case, it will be reflected by changing the icon that is displayed next to the crop. The same sequence is repeated when unselecting the crop.[6]

Figure 4-2. UI refresh with notifyListeners

- ***ChangeNotifierProvider*:** Widget that provides an instance of a *ChangeNotifier* to its descendants. For providing more than one class, as we do in our app, the `MultiProvider` widget is used. It is common to put this widget at the root of the app's widget tree so that all widgets which need it can access it. Below is an excerpt of the *main.dart* file of our app; here, the `MaterialApp` widget is wrapped inside a `MultiProvider`, so that the data of our different models is exposed to all widgets below.

```
void main() => runApp(MyApp());

class MyApp extends StatefulWidget
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext _) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (c) => CartModel()),
        ChangeNotifierProvider(create: (c) => CropListModel()),
        ChangeNotifierProvider(create: (c) => FertListModel()),
        ChangeNotifierProvider(create: (c) => ParamsModel()),
        ChangeNotifierProvider(create: (c) => ResultsListModel()),
        ChangeNotifierProvider(create: (c) => HistoryModel()),
      ],
      child: Builder(builder: (context) {
        return MaterialApp(
          title: 'Fertilicalc',
          theme: ThemeData(
...
```

Use of MultiProvider in *main.dart*

- ***Consumer*:** Widget that accesses the data from the *ChangeNotifier*. We must specify the type of model that we want to access; in our example, we want `CropListModel`, so we write `Consumer<CropListModel>`. This code accesses the model to build the list of crops on the screen. The excerpt below shows how each crop is built in the UI:

[6] In the final design of the app, tapping on an unselected crop will trigger the appearance of a Crop Dialog, instead of directly adding it (see section 5.2.2).

```
return Consumer<CropListModel>(
      builder: (context, list, child) => ListTile(
          leading: (list.items[index].isSelected
              //Shows 'Check' button if it is selected
              ? IconButton(
                  onPressed => list.toggleSelected(index);
                  icon: checkIcon,
                )
              //Shows 'Add' button that triggers dialog if not selected
              : IconButton(
                  onPressed => list.toggleSelected(index);
                  icon: addIcon,
                )),
```

Use of Consumer<CropListModel>

`builder` is a function that is called whenever the *ChangeNotifier* changes; when `notifyListeners()` is called, the `builder` methods of all the corresponding *Consumer* widgets are called. In our example, a ternary operator checks the value of the `isSelected` field to determine which icon to display for a specific crop (as illustrated in figure 4-1).

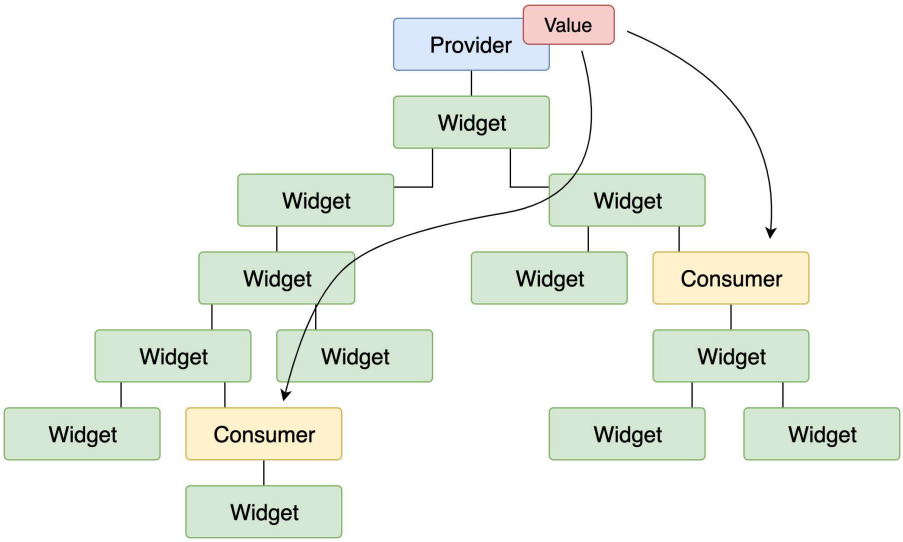


Figure 4-2. App State Management with Provider [25]

Essentially, Provider is a widget that makes some value – like the state of a *model* object – available to the widgets below it. A *Consumer* widget listens for changes in the value and rebuilds the widgets below itself when changes occur. [25]

There are additional ways to read a value from Provider; one we use extensively is the method `context.read<T>`, which returns `T` without listening to it. This method won't trigger widget rebuild when the value changes and can be called freely outside the `build` methods. [26]

We use this method, for instance, in our app's *commands* (see section 5.5), so that we can perform operations on the data contained by the models, from outside the widget tree.

Another advantage of Provider is that it can be examined through *DevTools*. *DevTools* is a suite of performance and debugging tools for Dart and Flutter, which allows, among other tasks, to inspect the UI layout and state of a Flutter app. The screenshot below shows the inspection of one of our models:

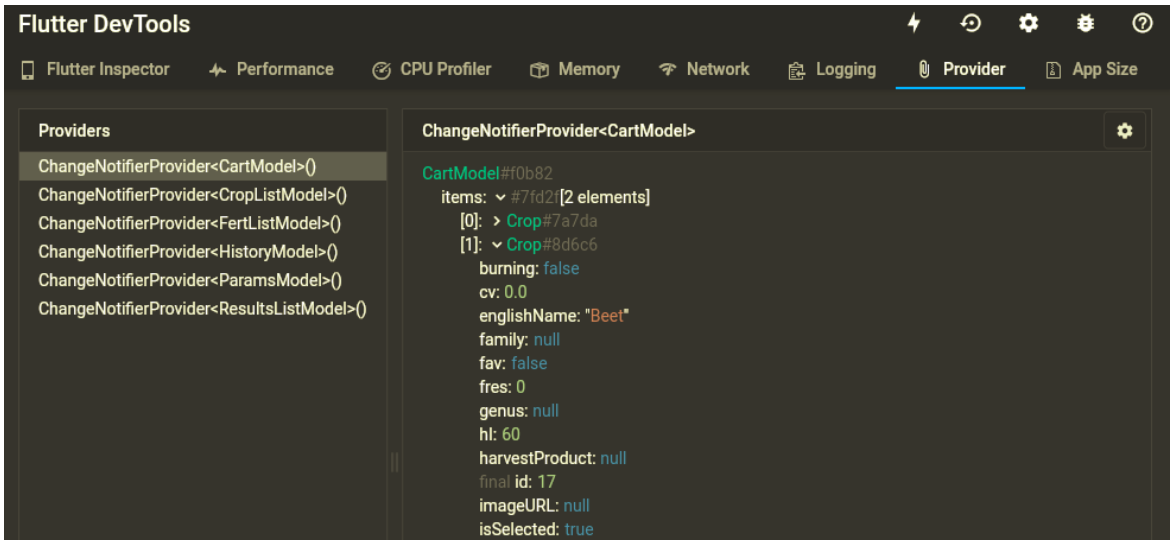


Figure 4-3. Inspection of Provider through DevTools

The *Providers* that appear on the left are the models that encapsulate the mutable data in our app, which may be displayed in the user interface. We have selected the `CartModel` object (see section 5.3.2), which in this instance contains a list of two `Crop` objects. We can see the different attributes of one of these crops; whenever the value of one of these attributes is changed, it will automatically be reflected in the user interface (as in figure 4-2).

4.3 App architecture

As a project gets more complex, the need for defining an architecture becomes essential to keep the code scalable and maintainable. At the end of developing the first prototype of our app, this necessity became imperative, and the code was refactored to accommodate a structured architecture.

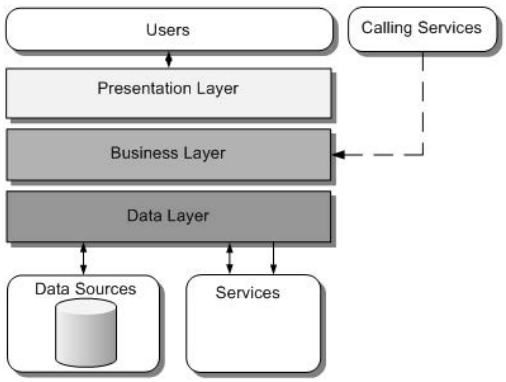


Figure 4-3. General mobile app architecture [27]

Mobile app architecture design usually consists of multiple layers, including these three fundamental ones:

- **Presentation Layer –** Contains UI components as well as the components processing them. Its focus is how to present the app to the end-user.
- **Business Layer –** Also commonly referred to as **Logic** Layer. It coordinates the application, processes commands, makes logical decisions and evaluations, and performs operations. It also moves and processes data between the two surrounding layers. [28]
- **Data Layer –** It encapsulates data persistence mechanisms and exposes the data within the mobile app.

It should provide an API to the business layer that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms.

The main advantage of using a multi-tiered architecture is that the different layers can be developed and maintained as independent modules, consequently improving maintainability and scalability. The goal is to keep the layers as independent as possible from one another; changing code in one layer should not affect the functionality of the others. Mobile app architecture patterns, such as the one we aim to use, serve to ensure that the code is clean, reliable, fast, and easy to maintain.

For our app, we have followed a *Model –View – Commands + Services* architecture.[7]

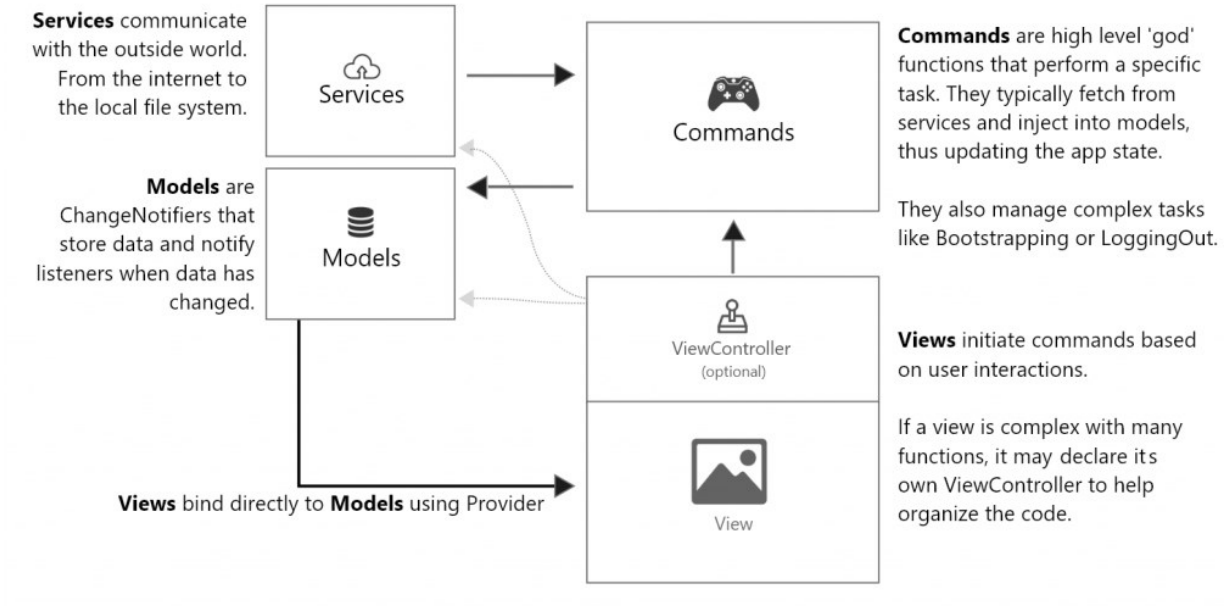


Figure 4-4. *Model – View – Commands + Services* architecture [29]

It divides most classes in our code into 4 tiers or layers:

- **Commands.** Application-level tasks (e.g., load data from memory) that are encapsulated in an object instance and started with a call (e.g., `LoadData().run()`). Commands can be initiated from the View layer, based on UI events, or via other Commands. Commands encapsulate their own state and can be asynchronous.
- **Services.** Tasks that interact with the "outside world", i.e., anything outside the application sandbox (Internet, the local file system, etc.). Additionally, they parse/return any data they receive. They never interact with the Model directly; instead, Commands will make Service calls, and potentially update the Model with the results.
- **Models.** Classes that hold the state of the application and encapsulate data, notifying listeners when data has changed. It provides an API to access, filter, and manipulate this data.
- **Views**. Widgets that are used to build the different screens within the application. Along with the UI code, it contains some basic logic to "control" the View. In some instances, the View communicates directly with the Provider to perform basic tasks. For instance, in the example seen in figure 4-2, the View calls the Model's `toggleSelected` method directly.

The crucial reason for choosing this architecture is that it takes full advantage of the *Provider* package as a state management approach and the easy data binding between Model and View that it offers. This allows us to simplify and modularize the code, improving its manageability and reusability.

[7] Our approach is heavily based on the article at [29]

The following are other advantages this architecture offers:

- Commands remove most of the business logic from the Model. By separating logic and state, scalability is improved.
- Commands are encapsulated; therefore, they never collide with other Commands and can store internal state easily. As opposed to a method from a Model, one can run any number of concurrent Commands.
- Commands are easily chain-able, allowing for increased flexibility when composing the business logic.
- Commands can access any combination of models and services to perform their functions. This removes most dependencies between models, making it easier to work with multiple data sets.
- The decoupling of services from commands improves the maintainability and testability because the commands do not depend on the concrete implementation of the interaction with external services. Commands simply call the methods exposed by the services.

The following diagram illustrates how the different modules interoperate inside our app:

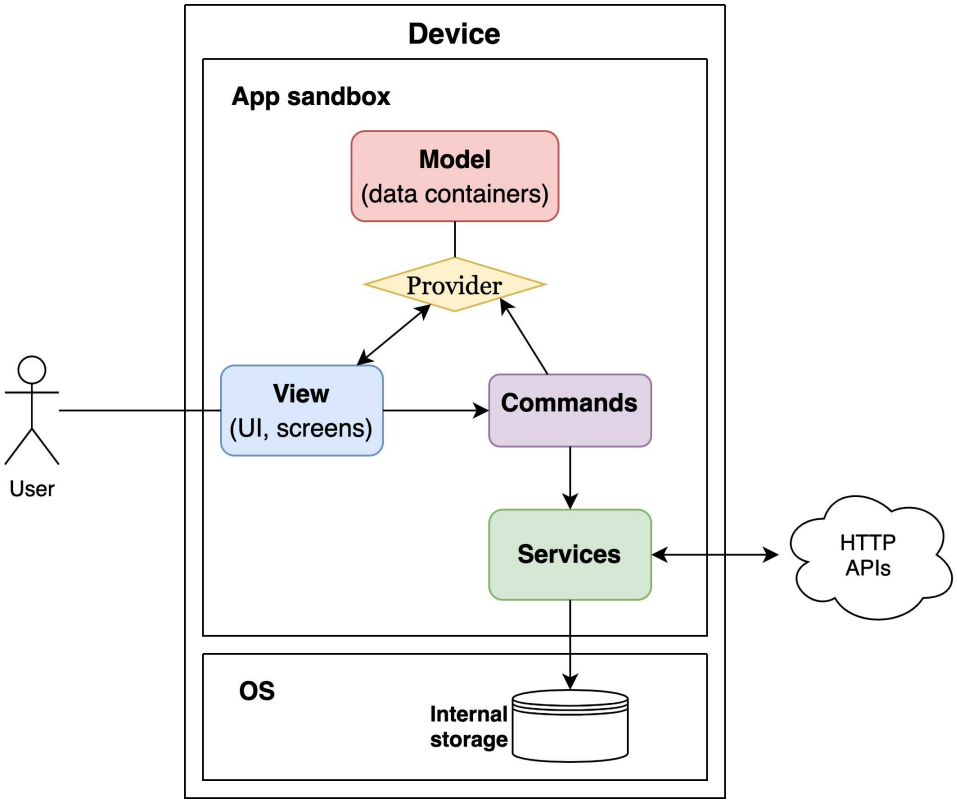


Figure 4-4. Integration of architectural components of the app

Here we see the central feature of our architecture: the use of Provider as the intermediary between Model and the other modules. When the View receives user input, it can change data from the Model by accessing the Provider, either directly, or indirectly through Commands. When the Model is updated, the View is directly updated through Provider. Services, which interact with external entities, are only accessed directly by Commands.

The implementation and integration of the different modules are further detailed in the following chapter.

5 Design and Implementation

The development of our app began with the production of an initial prototype that implemented only critical requirements. The ease of use of the Flutter framework to quickly build UIs from scratch makes it attractive to employ an iterative style of development; taking advantage of this, more requirements were gradually added, and the structure was adapted to the architecture presented in section 4.3. The evolution from the first screen mockups to the final design is illustrated in Appendix B.

In this chapter, we examine the final design of our app, how the different modules from our architecture are implemented, and how they are integrated to fulfill the requirements of our app.

5.1 Application flow

The basis of our design is the following application flow, which illustrates the journey of the user through the different screens of the app:

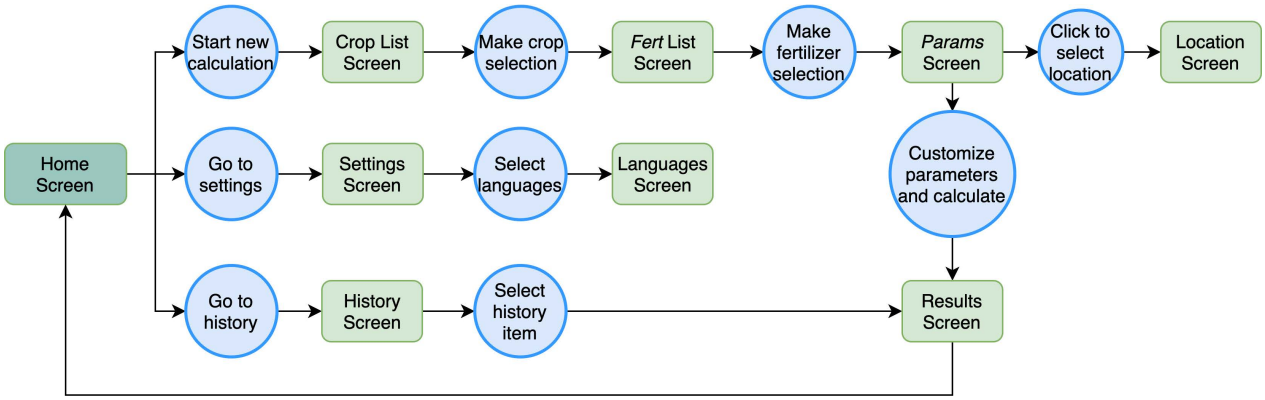


Figure 5-1. Overview of the application flow

The starting point of the user flow is the Home Screen; from here, three different paths can be taken:

- Start of a **new calculation**.
  - The app first displays the 'Crop List Screen'.
  - Once the user makes a crop selection, the app navigates to the '*Fert* List Screen'
  - Once a fertilizer selection is made, the app navigates to the '*Params* Screen'. When the user chooses to select a location, the app navigates to the 'Location Screen' and back.
  - Once the user customizes the global parameters, he can trigger the calculation of results, after which the app navigates to the 'Results Screen'.
- **Settings** section of the app.
  - The app first displays the 'Settings Screen'.

  - If the user chooses to change the language, he is directed to the ‘Languages Screen’.
- **History** section of the app.
  - The app first displays the ‘History Screen’.
  - If the user selects any of the items in history, he is directed to the ‘Results Screen’, which is reused from the calculation path.

Note that all navigation in the app is reversible; at any screen, the user can go back to the previous screen.

5.2 View

In this section, we discuss how the different screens are implemented through the View module and show how the app’s screens look on the device. For the sake of brevity, we only mention certain elements of the View to illustrate how the different tools provided by the Flutter framework are used and how the proposed architecture is applied.

The View encompasses code that describes the presentation of the UI: how it should look, what data it should display, how it should interact with the user, etc. It also includes some code that bridges the gap between the presentation and the business logic. For this purpose, it contains functions that perform tasks such as calling the commands to perform more complex tasks, navigating to other screens, customizing search results based on user input, etc.

5.2.1 Home Screen

The Home Screen is the starting point of the application flow. It is built as a stateful widget, which allows the screen to trigger its own rebuilding in response to changes in the state.

```
class HomeScreen extends StatefulWidget {
  HomeScreen({Key key}) : super(key: key);

  @override
  _HomeState createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addPostFrameCallback((_) async => await _loadData());
  }

  Future<void> _loadData() async {
    await InitDataLoad().run();         //Loads initial data
    await LoadFavs().run();             //Loads favs (crops and ferts)
    await LoadLanguage().run();         //Loads language for the whole app
  }

  @override
  Widget build(BuildContext context) {
    return SafeArea(
      child: Scaffold(
          bottomNavigationBar: ConvexAppBar(
              ...
              items: [
                TabItem(
                    icon: Icons.history,
                    title: AppLocalizations.of(context).layoutValue("History")),
                TabItem(icon: Icons.add),
                TabItem(icon: Icons.settings, title:
                          AppLocalizations.of(context).layoutValue("Settings")),
              ],
              onTap: (int i) {
                if (i == 0) Navigator.pushNamed(context, '/history');
```

```
            //If we return, we unselect any crops and ferts
            if (i == 1)
              Navigator.pushNamed(context, '/cropList')
                  .whenComplete(() async => await ProviderCommand().clearCart ());
            if (i == 2) Navigator.pushNamed(context, '/settings');
          }),
        body: Container(
            decoration: BoxDecoration(
        ...
```

Home Screen

Before loading the view, we perform some initializations inside the `initState` method, which is called when first building a widget. We call the `_loadData` function, which in turn calls the appropriate commands to load initial data, load the favorites selections, and load the language in which the app should be displayed.

The `build` method is in charge of building the screen. The `SafeArea` widget ensures there is sufficient padding to avoid intrusions by the operating system, i.e., avoiding issues with notches. The `Scaffold` widget implements the basic *Material Design* layout structure.

The bottom navigation bar is implemented using the third-party widget `ConvexAppBar` [30], which defines the appearance (`items` property) and behavior (`onTap` property) of the three buttons at the bottom of the screen. To navigate to the different screens, we use the `Navigator` tool provided by the Flutter framework [31].

The figure below illustrates the navigation paths that stem from the Home Screen:

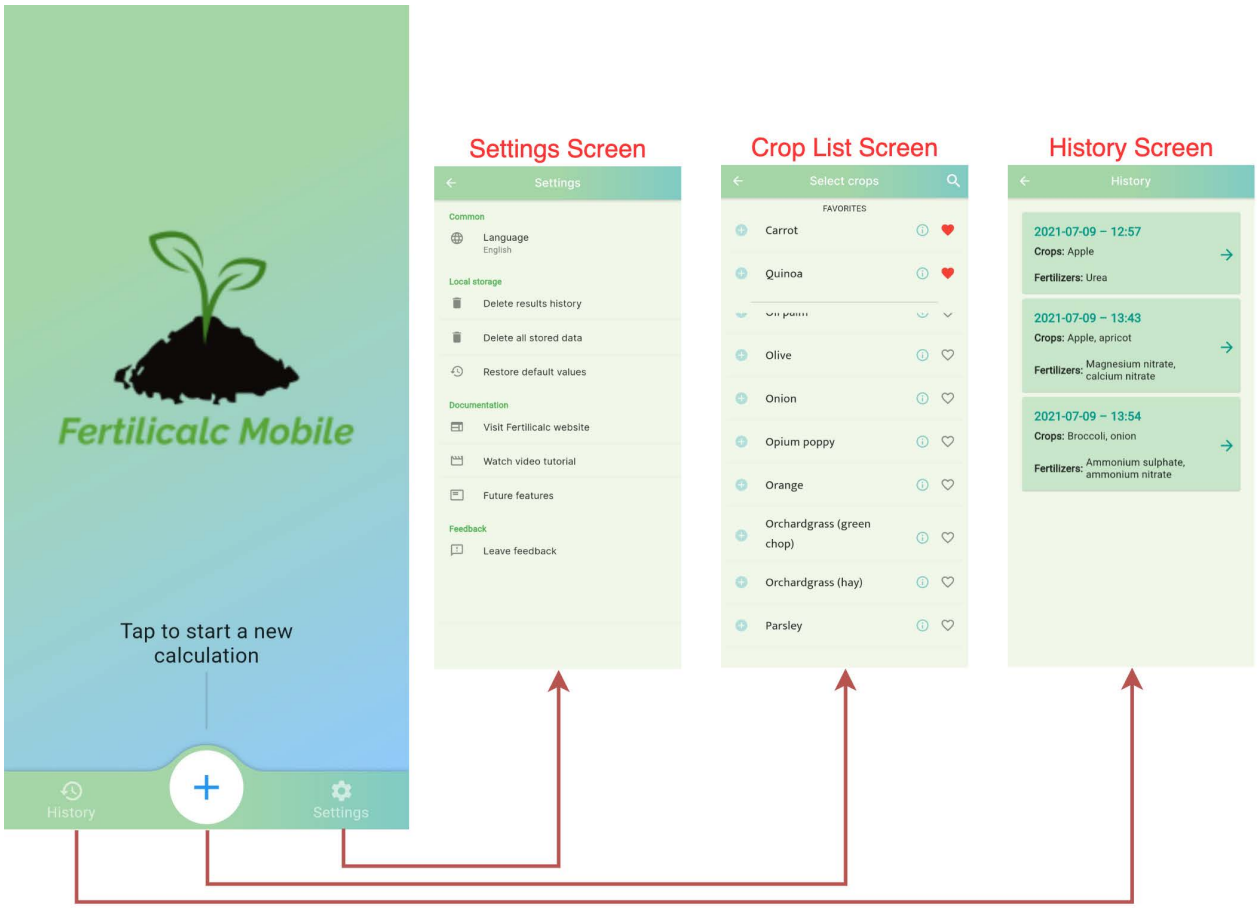


Figure 5-2. Navigation from Home Screen

5.2.2 Crop List Screen

The Crop List Screen is defined mainly through three widgets: `CropListScreen`, `CropDialog`, and `CropInfoSheet`. The `CropListScreen` widget builds the complete screen; the latter two define additional

components that are integrated with the screen. This division was made for better maintainability, taking advantage of the modularity of widgets.

The following is the first part of the `CropListScreen` code:

```dart
import 'package:provider/provider.dart';
•••

List<Crop> cropList = [];      //For search

class CropListScreen extends StatefulWidget {
  •••
}

class _CropListScreenState extends State<CropListScreen> {
  //Navigates forward in the user flow
  void navForward() async {
    FavsMgmt().saveCropFavs();                //Saves the favs
    PrepareCart().run();                      //Updates the cart with the chosen crops
    Navigator.pushNamed(context, '/fertList'); //Navigates to Fert List Screen
  }

  @override
  Widget build(BuildContext context) {
    return SafeArea(child: Consumer<CropListModel>(builder: (context, list, child) {
      return Scaffold(
          appBar: CustomAppBar(
              AppLocalizations.of(context).layoutValue('Select crops'),
              IconButton(
                  icon: searchIcon,
                  onPressed: () {
                    cropList = context.read<CropListModel>().items; //For search
                    showSearch(context: context, delegate: DataSearch());
                  })),
          bottomNavigationBar:
              list.items.where((element) => element.isSelected == true).isNotEmpty
                  ? CustomFloatingButton(() => navForward(),
                      AppLocalizations.of(context).layoutValue('Select fertilizers'))
                  : SizedBox.shrink(),
          body: cropListBody(list));
    }));
  }

  Widget cropListBody(CropListModel list) {
    //Counts the number of favorites
    var _favCount = list.items.where((element) => element.show == false).length;

    List<Crop> _favItems;          //Defines list of favorites
    if (_favCount > 0)
        _favItems = list.items.where((element) => element.show == false).toList();

    //Builds the column with favorites list + crops list
    return Padding(
      padding: const EdgeInsets.all(10),
      child: Column(
        children: [
          //Only builds favorites list if there are favorites
          (_favCount > 0)
              ? Column(
                  children: [
                    Text(AppLocalizations.of(context).layoutValue('favorites').toString(),
                    SizedBox(
                      height: (_favCount * 70).toDouble(),
                      child: ListView.separated(
                          separatorBuilder: (context, index) { return Divider();   }
                          shrinkWrap: true,
                          itemCount: _favCount,
                          itemBuilder: (context, index) {
                            return CustomCropTile(context, _favItems[index].id - 1, 0);
                          }),
                    ),
                    Divider(thickness: 2, indent: 40, endIndent: 40),
                    SizedBox(height: 10)
                  ],
                )
              : SizedBox.shrink(),
```

```
72          //Builds the crop list
73          Expanded(
74            child: ListView.separated(
75                separatorBuilder: (context, index) {
76                  return list.items[index].show ? Divider() : SizedBox.shrink();
77                },
78                shrinkWrap: true,
79                itemCount: list.items.length,
80                itemBuilder: (context, index) {
81                  //Hides favorites from the general crop list
82                  return list.items[index].show
83                      ? CustomCropTile(context, index, 0)
84                      : SizedBox.shrink();
85                }),    ],           ),       );      }
```

Crop List Screen (I)

We have nested most of the widget tree inside a `Consumer` (line 20). This makes it more convenient to read the data from the `CropListModel`.

Inside the `Scaffold`, we define an app bar (line 22) by instantiating `CustomAppBar`, which is defined in another file and reused in other screens. It takes two parameters: a title and a trailing widget, i.e., some widget that will appear to the right of the title. In this case, it is a search icon button that will trigger the start of a search. The `onPressed` property of the `IconButton` (line 26) is a call to the `showSearch` function, which we will define shortly.

For the bottom navigation bar (line 30), we define a floating button (also an auxiliary widget, `CustomFloatingButton`), that will appear when we have selected at least one crop and will call `navForward` when pressed. This fulfills requirement R-08.

The `navForward` function (line 12) defines what the app should do when navigating to the next screen: save the chosen favorites, update the cart with the chosen crops, and navigate to the next screen. The first two are done by calling the appropriate commands; the last by using the framework's `Navigator` tool.

Throughout this screen and other screens, the translation of various texts is done by using the `AppLocalizations` tool (lines 23, 33, 56).

For building the body (line 35), we define the `cropListBody` widget, passing as a parameter the list of crops obtained from the Provider. This widget builds a section with the list of favorites (line 54) and a section with the general list of crops (line 73). For both, it uses the `ListView.separated` constructor provided by the Flutter framework [31].

To build each element from the list, we define the `CustomCropTile` widget:

```
86  Widget CustomCropTile(BuildContext context, int index, int queryLength) {
87    return Consumer<CropListModel>(
88        builder: (context, list, child) => ListTile(
89            contentPadding: const EdgeInsets.only(),
90            leading: (list.items[index].isSelected
91                //Shows 'Check' button if it is selected
92                ? IconButton(
93                    onPressed: () {
94                      list.toggleSelected(index);
95                      list.setY(index, 0);
96                    },
97                    icon: checkIcon,
98                  )
99                //Shows 'Add' button that triggers dialog if not selected
100               : IconButton(
101                   onPressed: () => showMyDialog(context, index),
102                   icon: addIcon,
103                 )),
104           title: Material(
105               color: Theme.of(context).scaffoldBackgroundColor,
106               child: InkWell(
107                   //Triggers dialog if it is already selected
108                   onTap: () {
109                     if (list.items[index].isSelected) showMyDialog(context, index);
110                   },
```

```
111                   //Shows crop name with special styling for search
112                   child: queryLength > 0
113                       ? RichText(
114                           text: TextSpan(
115                               text: list.items[index].name.toString().substring(0, queryLength),
116                               style: defaultBoldStyle(20),
117                               children: [
118                               TextSpan(
119                                   text: list.items[index].name.toString().substring(queryLength),
120                                   style: defaultGreyStyle(20))
121                             ]))
122                       : Text(AppLocalizations.of(context).cropName(index),
123                           style: defaultTextStyle(20)))),
124           trailing:
125         ...
126         [       //'Info' button that navigates to Crop Info Screen
127                   IconButton(
128                       onPressed: () => _showCropInfoSheet(context, index),
129                       icon: Icon(Icons.info_outline, color: Colors.teal[200])),
130                 ),
131                 //'Fav' button
132                   IconButton(
133                       icon: list.items[index].fav ? favoriteIcon : favoriteBorderIcon,
134                       onPressed: () => list.toggleFav(index)),
135         ],      ),          )));       }
```

CustomCropTile – Crop List Screen (II)

This widget is defined outside the state's scope because it is also called by the search functions, which are defined in another class. Therefore, one of the arguments it takes is a `BuildContext`, which is needed to access the Provider. It also receives the crop's index, which it uses to retrieve the crop's info from the Provider, and `queryLength` (length of the user's search query), which is used to style the search results.

When a crop has not yet been selected and the user presses its 'Add' button, the `showMyDialog` function is called, passing it the crop's index (line 109). This triggers the appearance of the `CropDialog` widget (which we will define shortly).

When the 'Info' button is pressed, the `_showCropInfoSheet` function is called. This displays a bottom sheet, defined in `CropInfoSheet`, which shows additional crop info.

The following is the `DataSearch` class, which implements the search functionality (R-04):

```
136 class DataSearch extends SearchDelegate<String> {
137   final crops = cropList;
138   List<Crop> suggestionList = [];
139
140         ...
141
142   @override
143   Widget buildResults(BuildContext context) {
144     //Selects crops that fit the current search query
145     suggestionList = crops
146         .where((p) => p.name.startsWith(new RegExp(query, caseSensitive: false))).toList();
147
148     return Padding(
149         padding: const EdgeInsets.fromLTRB(30, 10, 20, 10),
150         //Builds a list of all the crops returned by the search query
151         child: ListView.separated(
152             separatorBuilder: (context, index) => Divider(),
153             shrinkWrap: true,
154             itemCount: suggestionList.length,
155             itemBuilder: (context, index) {
156                 return CustomCropTile(context, index, query.length);
157         ...
```

DataSearch class – CropListScreen (III)

The following screenshots show the appearance of the screen on the device. On the right is the screen when the user is searching for a crop:

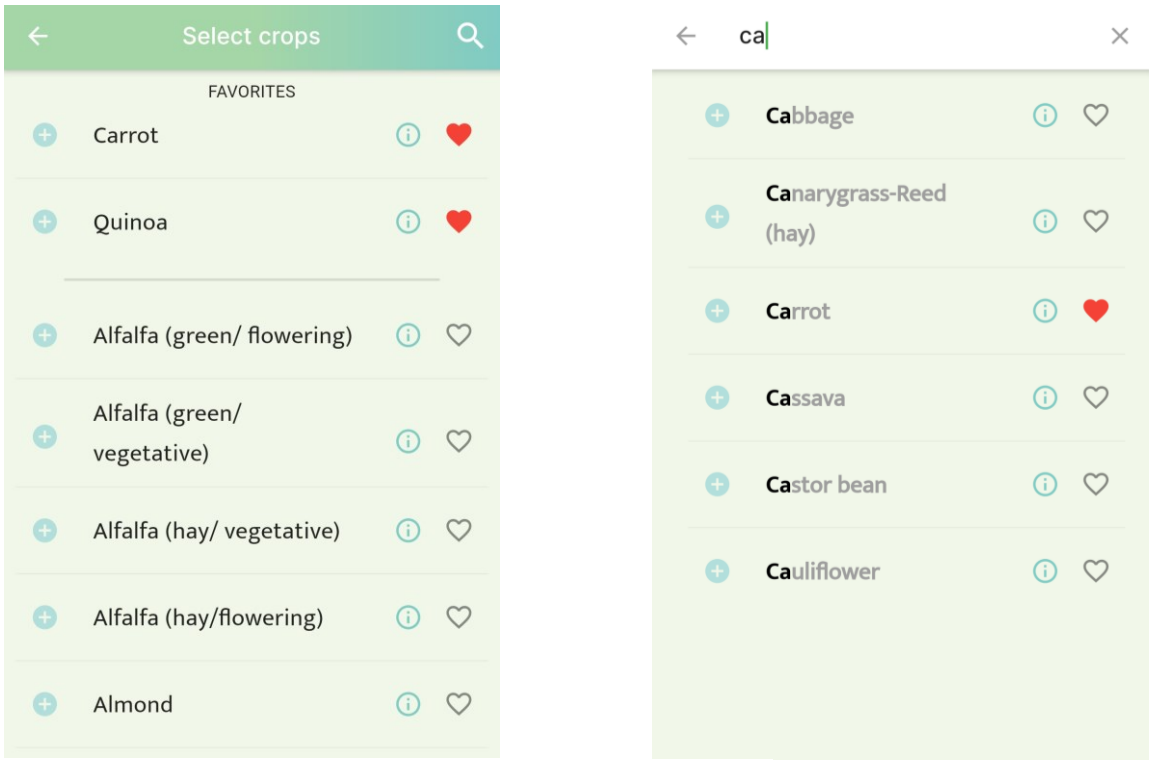


Figure 5-3. Crop List Screen

Below is the widget tree for a `CustomCropTile`, where we can observe the nesting of our custom widgets, and a screenshot which illustrates the layout of these widgets. The `cropListBody` contains the `ListView` that, using its `ListView.separated` builder, builds the list of `CustomCropTile` – `Divider` pairs.

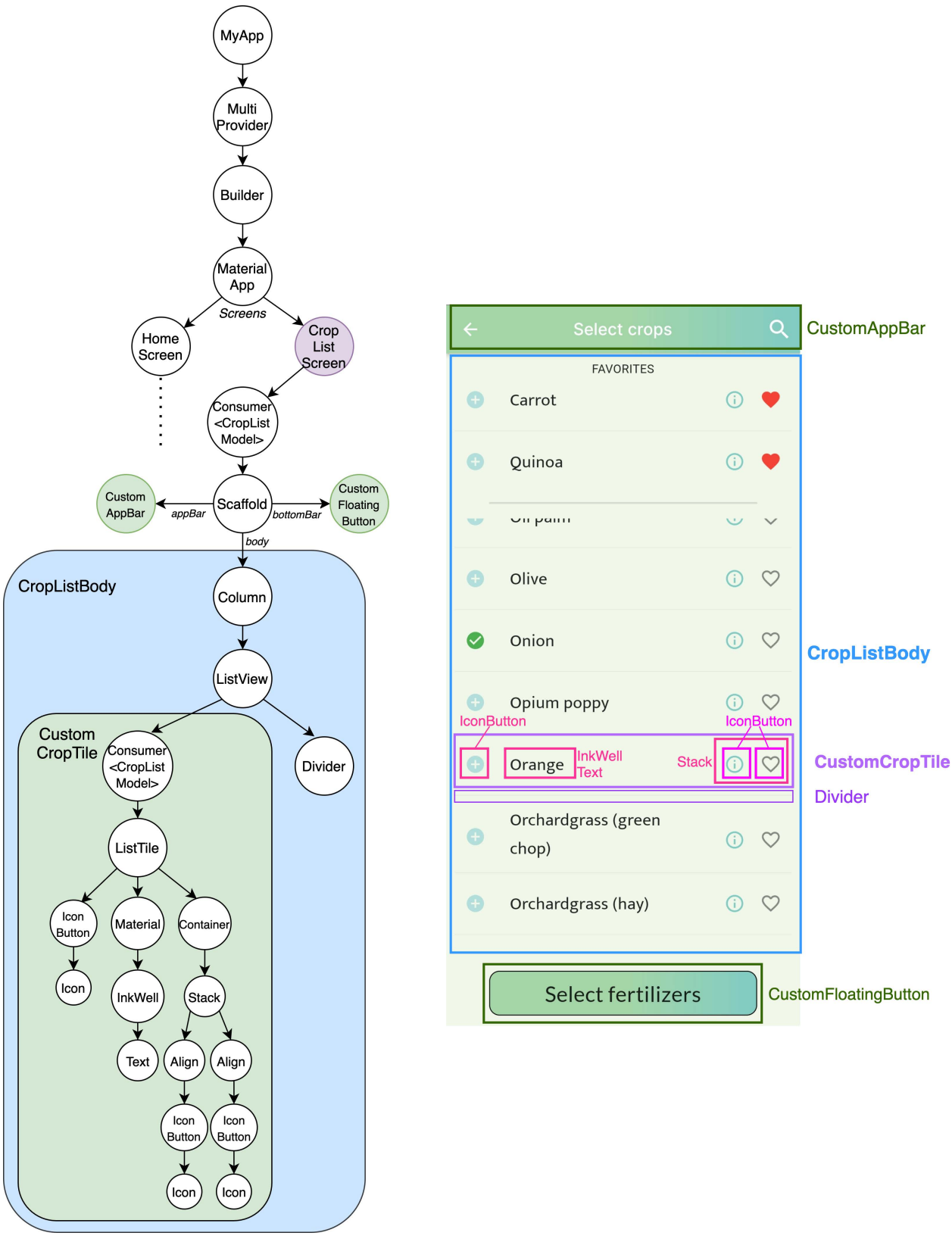


Figure 5-4. Widget tree of a CustomCropTile

The `CropDialog` is built using the framework's `showDialog<void>` method. It allows the user to enter a yield for the crop and edit the two additional parameters defined in R-03. For Boolean parameters, such as the 'Burning residues' one, a toggle-switch button is used. When the user enters a positive yield, the crop can be added by pressing the 'Accept' button, which fulfills R-02.

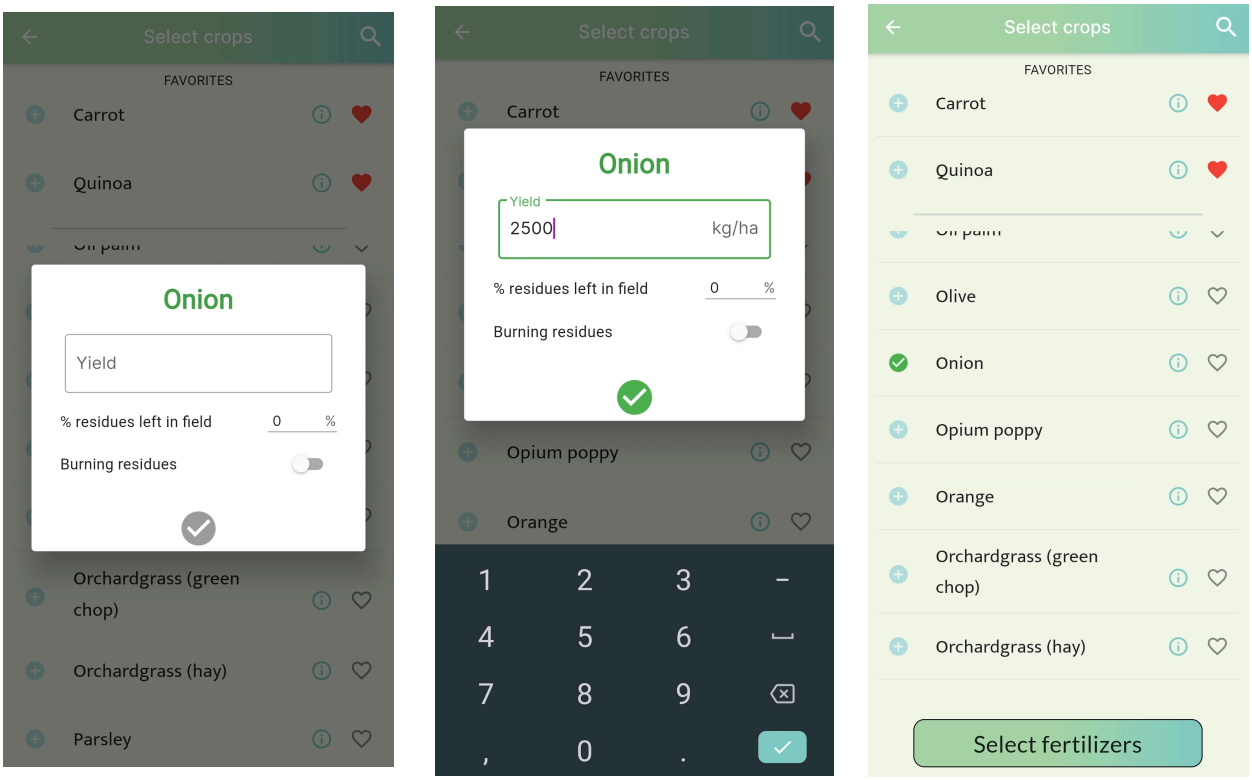


Figure 5-5. Addition of a crop through Crop Dialog

The `CropInfoSheet` is built using the framework's `showModalBottomSheet<void>` method. For a given crop, it shows its species name, genus, family, and a picture of the crop, fulfilling R-06.

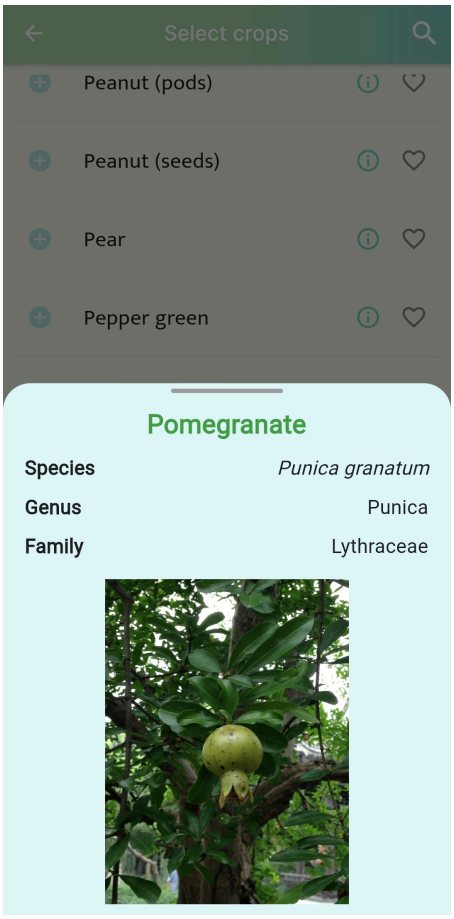


Figure 5-6. Crop Info Sheet

5.2.3 Fert List Screen

The Fert List Screen is built analogously to the Crop List Screen because the commands and data structures it interacts with are majorly similar. In the following screenshots we can observe its features:

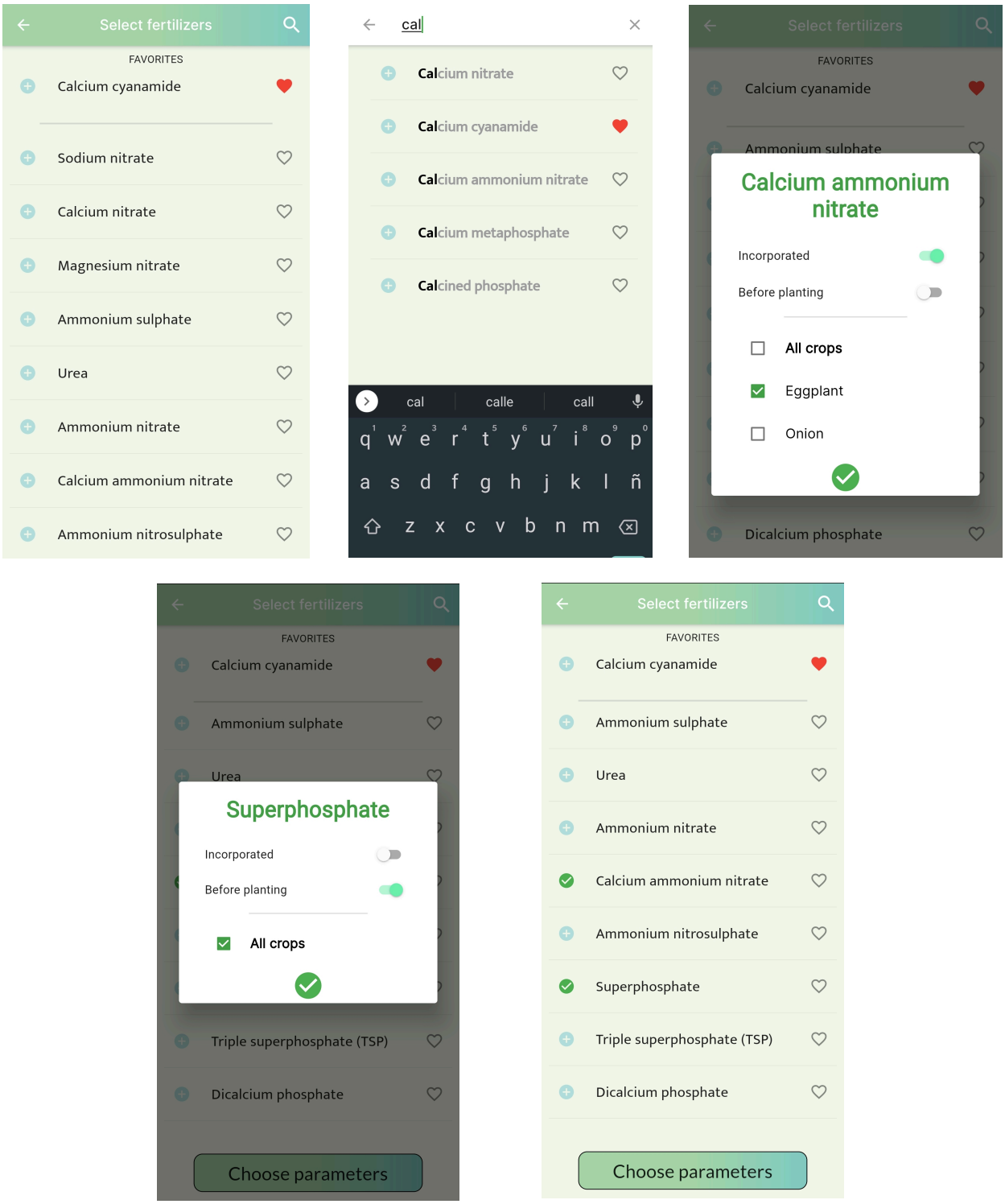


Figure 5-7. Fert List Screen

- The user can scroll through a list of the available fertilizers. (R-07)
- The user can mark a fertilizer as a favorite, then unmark it. (R-12) The list of favorites is displayed before the list of non-favorites.
- Pressing on the 'Search' button triggers the appearance of a search bar which allows the user to search for a specific fertilizer. (R-11)
- When the user presses on a specific fertilizer, a dialog pops up. This dialog allows the user to mark the

fertilizer as '*incorporated*' and/or '*before planting*' and choose which crops from the rotation to apply the fertilizer to. There is also the option to apply it to all crops in the rotation. (R-09, R-10)

- Once one or more fertilizers have been selected, the user can navigate to the next screen by tapping on the 'Choose parameters' button.

5.2.4 Params Screen

The Params Screen allows the user to choose his location and customize the global parameters of the calculation, fulfilling R-13 and R-15. The topmost item on the screen's body shows either a prompt asking the user to choose a new location (see figure below) or the last location chosen by the user (see figure 5-19). The rest of the body is a scrollable list of name-value pairs for each global parameter that the user can edit.

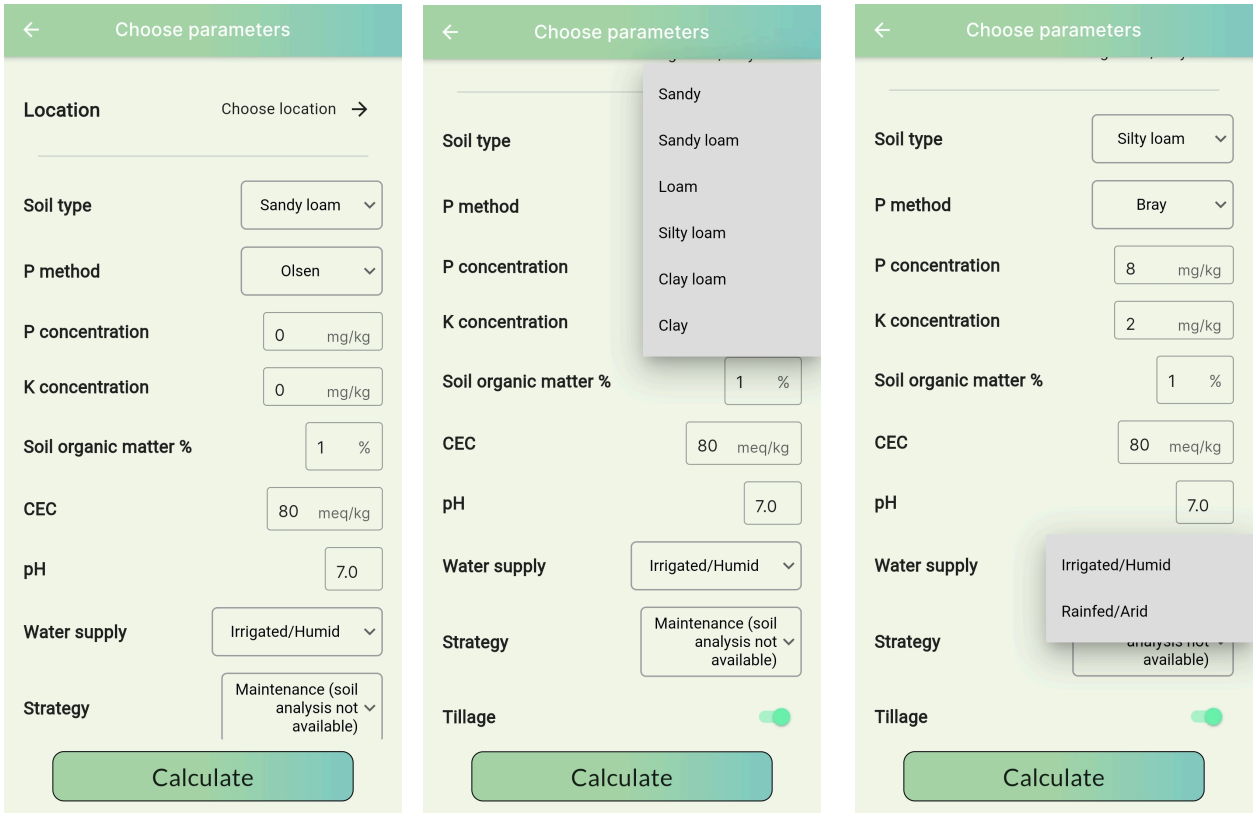


Figure 5-8. Params Screen

When a user asks to choose a new location, the screen invokes the *Permission_handler* package, which takes care of asking for the user's location permissions, adapting to both Android and iOS. [32] Only if the app detects the user has given these permissions, the user is sent to the Location Screen.

For updating each parameter, the corresponding dropdown button (e.g., *soil type*), input field (e.g., *P concentration*), or toggle-switch button (*tillage*) updates the Model directly when the user changes its value.

5.2.5 Location Screen

The Location Screen implements the *Google Maps Place Picker* package, which presents a Google Maps-like interface where the user can see his current location and pick any point in the world map. [33] The user can also search for a specific place through a search bar. The following is an excerpt of its code:

```
body: PlacePicker(
        apiKey: "…" // Key used to access Google Maps API
        onPlacePicked: (result) {
          var _latLng = LatLng(result.geometry.location.lat, result.geometry.location.lng);
          Provider.of<ParamsModel>(context, listen: false)
              .updateLocation(_latLng, result.formattedAddress);
          SaveLocation.run(result.geometry.location.lat,
              result.geometry.location.lng, result.formattedAddress);
          Navigator.of(context).pop(true);
                ...
```

PlacePicker widget

The `PlacePicker` widget, defined by the aforementioned package, communicates directly with the Google Maps API to provide all the necessary functionality.

When the user chooses a location, it returns an object that contains information such as the latitude and longitude, the address, etc. With this information, we update the location in the `ParamsModel` through the auxiliary method `updateLocation` (defined inside the model). We also store the location in the device's internal memory by calling the `SaveLocation` command. In both cases, we only store the coordinates and an address.

We have used a third-party package to outsource this functionality because it would be very complex to do it from scratch, especially since it involves interaction with both the Android and iOS underlying frameworks. This approach, however, comes with some limitations, since not many customizations can be made to the solution provided by the package, and it may be, at some point, deprecated.

Below is how the Location Screen looks on the device:

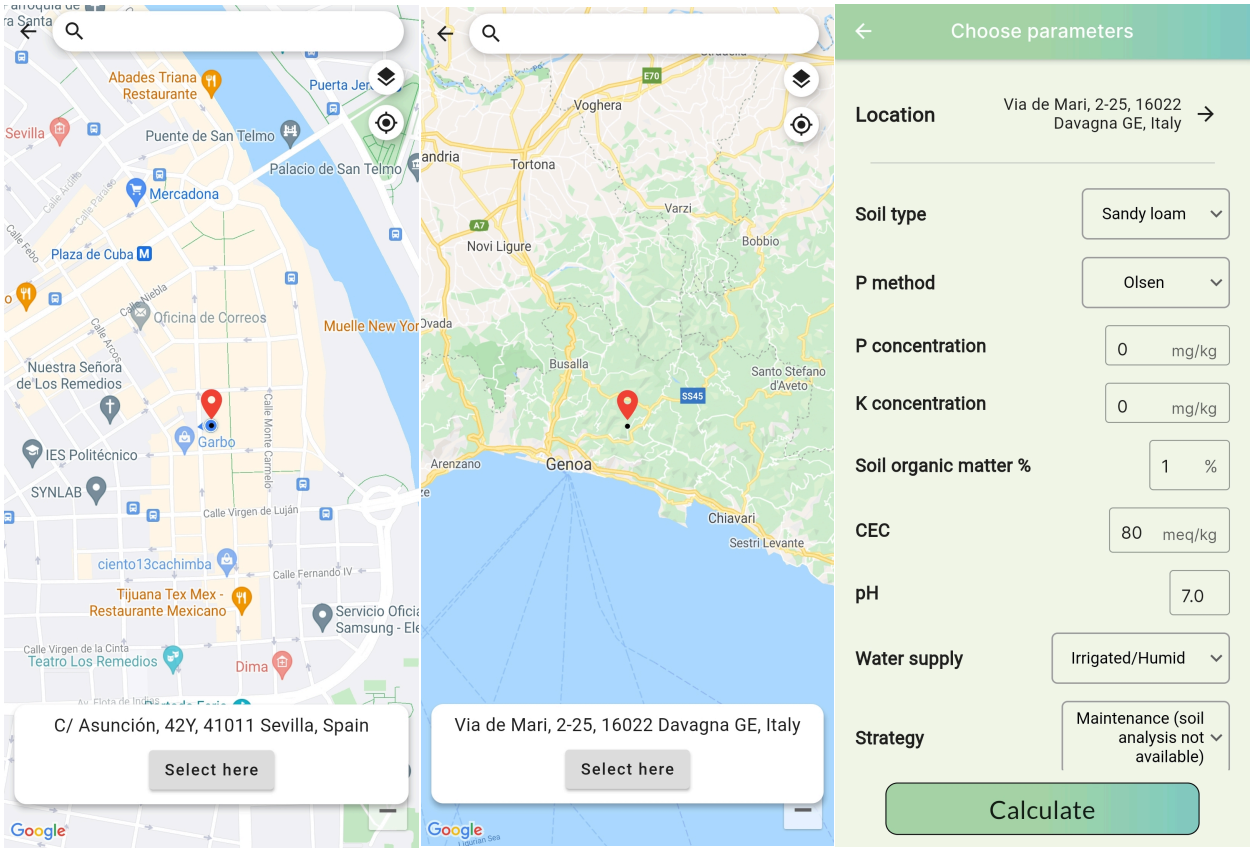


Figure 5-9. Location Screen

5.2.6 Results Screen

The Results Screen displays a card for each crop involved in the calculation. Each card contains the results for the corresponding crop (R-20). Initially, the card only shows the yield for the crop and the rates for each fertilizer. When the user taps the 'Expand' button, the card expands to show the rest of the parameters defined in R-20.

In the bottom bar of the screen, the user can choose to save the results (R-21) or share them with a third party (R-24).

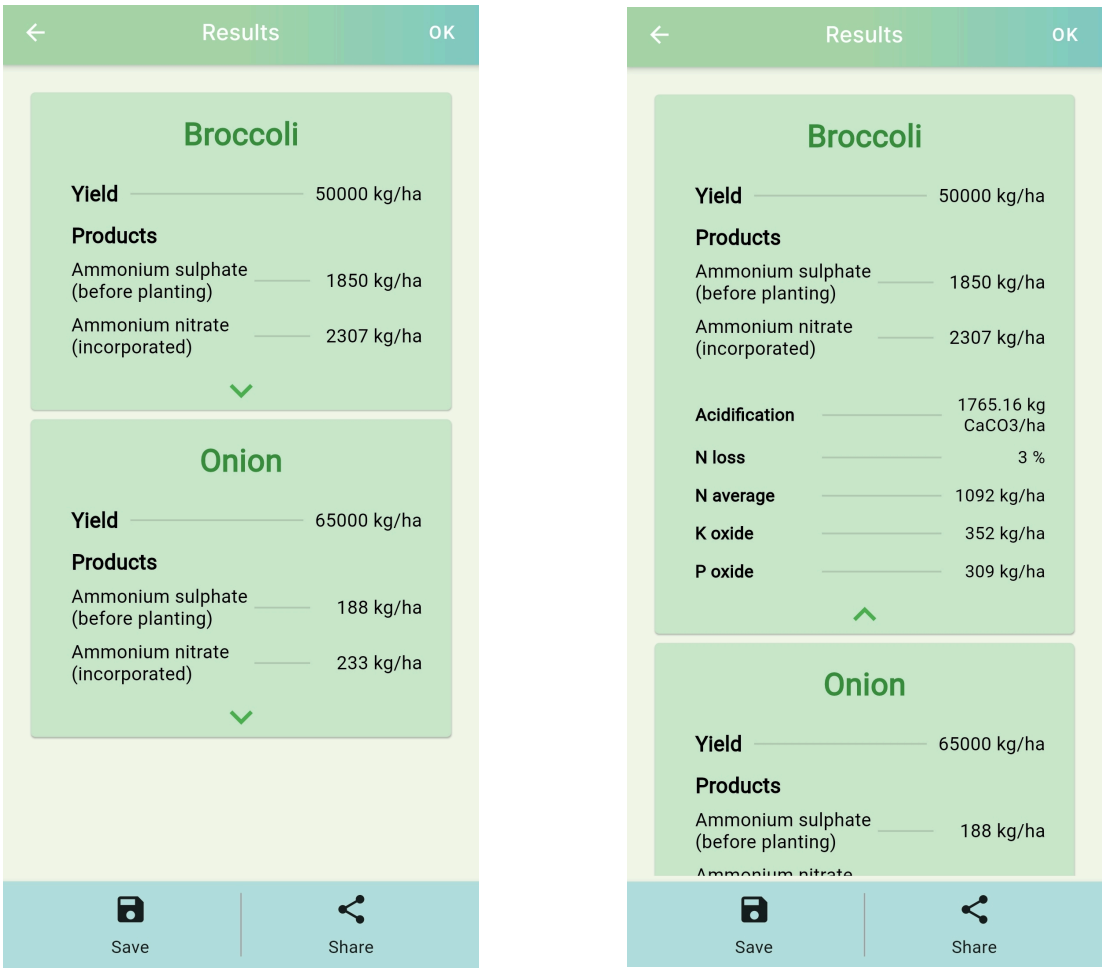


Figure 5-10. Results Screen (I)

5.2.7 History Screen

The History Screen displays a list of all the calculations that the user has saved (R-22). Each calculation is presented in a card with its date and time, and crops and fertilizers involved. Tapping on one of them makes the app navigate to the Results Screen, which displays the results for the chosen calculation (R-23).

The Results Screen displayed in this case only differs in the bottom bar, where instead of the option to save, the user has the option to delete the set of results (R-25).

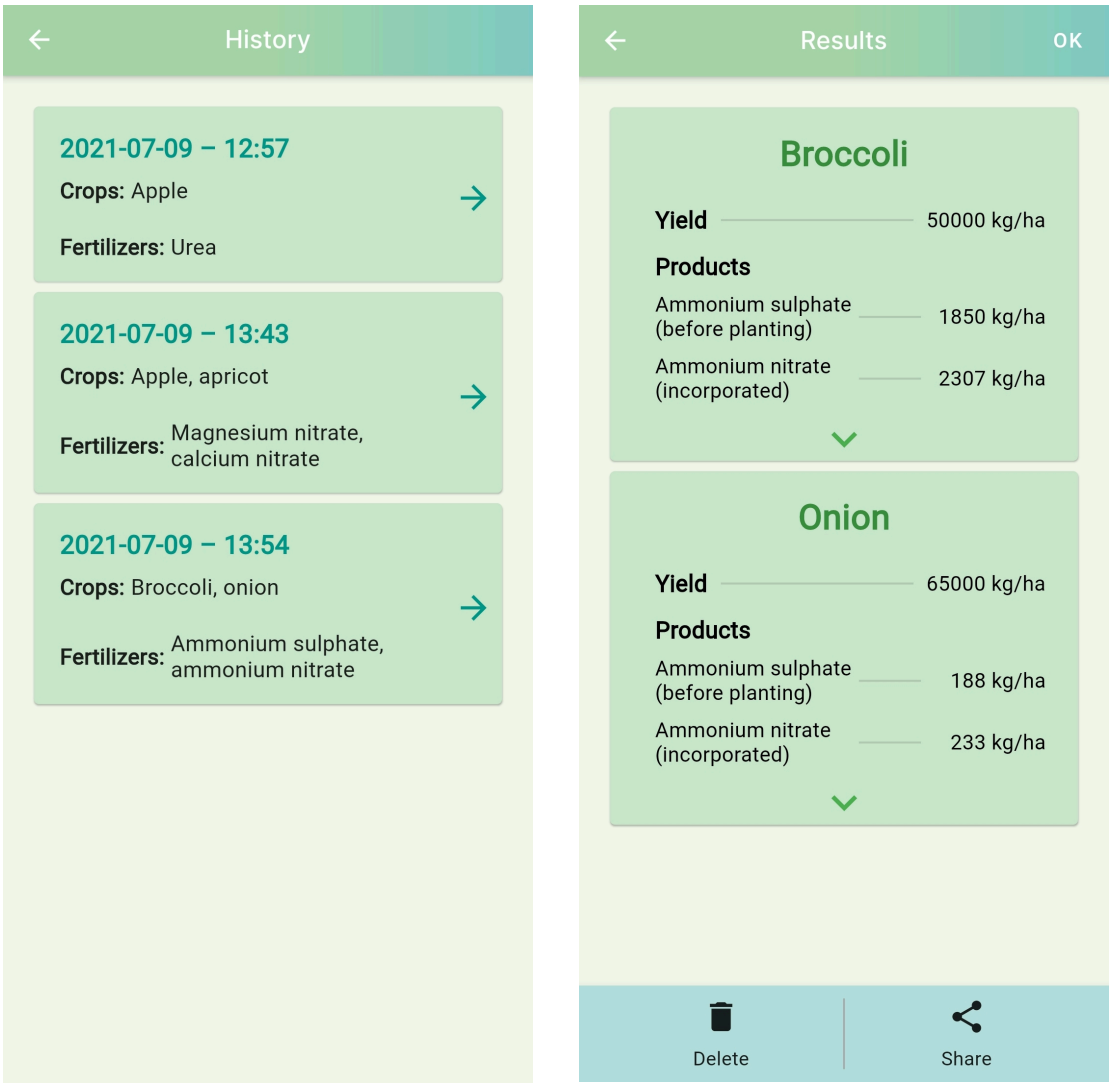


Figure 5-11. History Screen & Results Screen (II)

5.2.8 Settings Screen & Languages Screen

The Settings Screen uses the *Settings_UI* package [34] to display a list of sections, which we define as:

- **Common.** Includes the Language tile, which shows the currently chosen language and redirects the user to the Languages Screen.
- **Local storage.** Includes tiles that perform tasks related to the persistence of the user's data. The '*Delete results history*' tile calls the `deleteHistory` command (R-26); the '*Delete all stored data*' tile calls the `deleteAllData` command (R-27); and the '*Restore default values*' tile calls the `restoreDefault` command (R-17).
- **Documentation.** Includes tiles that provide the user with additional information to better understand the app. The '*Visit Fertilicalc website*' tile redirects the user to the official Fertilicalc website, the '*Watch video tutorial*' tile redirects the user to a Youtube video tutorial[8], and the '*Future features*' tile shows a dialog containing a list of improvements soon to be implemented.
- **Feedback.** The '*Leave Feedback*' tile redirects the user to a Google Forms page used for this purpose.

[8] We have elaborated a video tutorial for the app in English (https://youtu.be/KBc7tqCvHCI) and in Spanish (https://youtu.be/zTfh5sAfrhY).

Redirection to external websites is made using the `url_launcher` package [35].

The Languages Screen displays the list of languages available for the app. When the user changes the chosen language, this screen triggers the change of language and the storage of the user's choice by calling the appropriate commands.

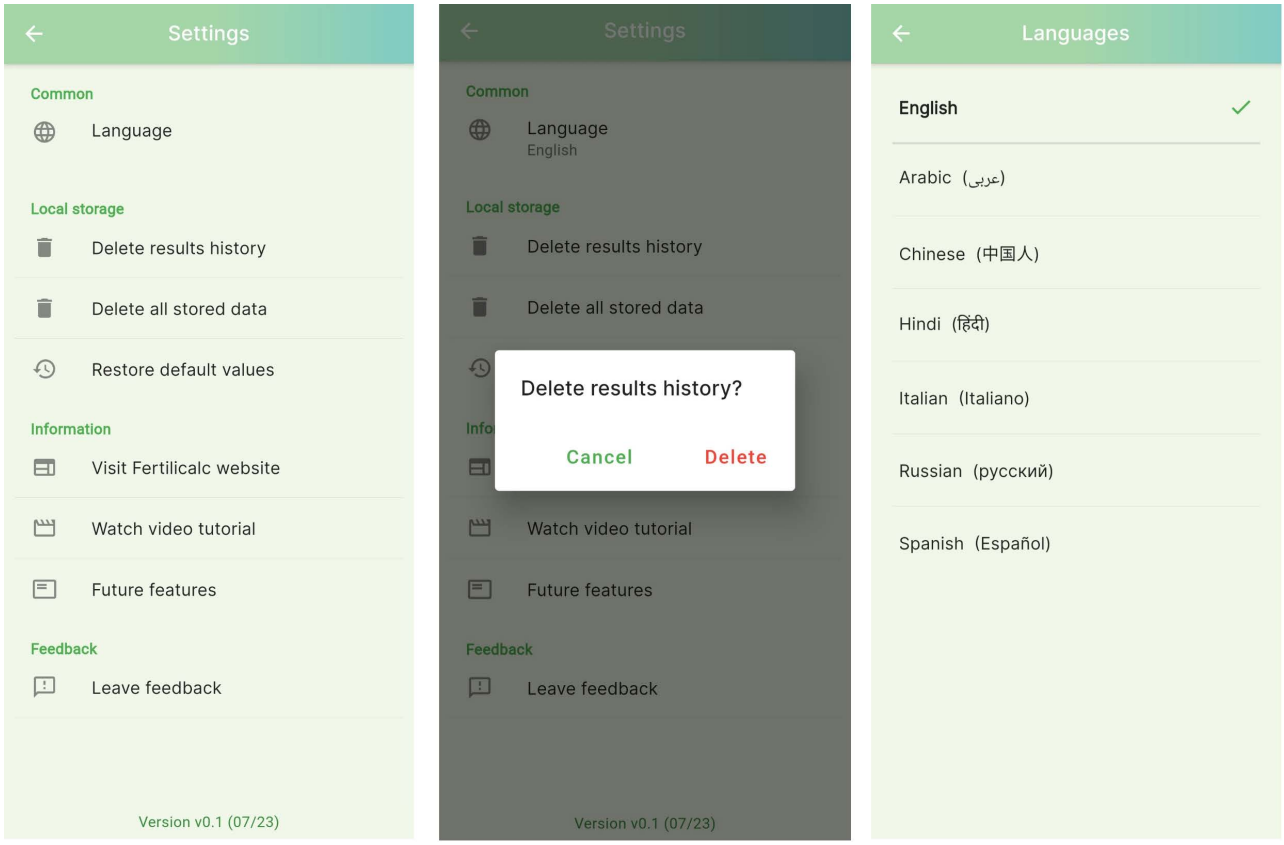


Figure 5-12. Settings Screen & Languages Screen

5.3 Model

The components of the Model module of the app encapsulate most of the data which is displayed on the screens. They mainly include two types of classes: normal data objects and change notifiers. The former are simple classes that just encapsulate objects, e.g., a crop along with all its properties. The latter store lists of objects, such as crops, along with additional data, and define some basic functions to filter and manipulate this data. They extend `ChangeNotifier` so that they can notify listeners when their data has changed, through the Provider mechanism outlined in section 4.2.

Below is the class diagram of the Model classes, where the *ChangeNotifier* classes are outlined in color:

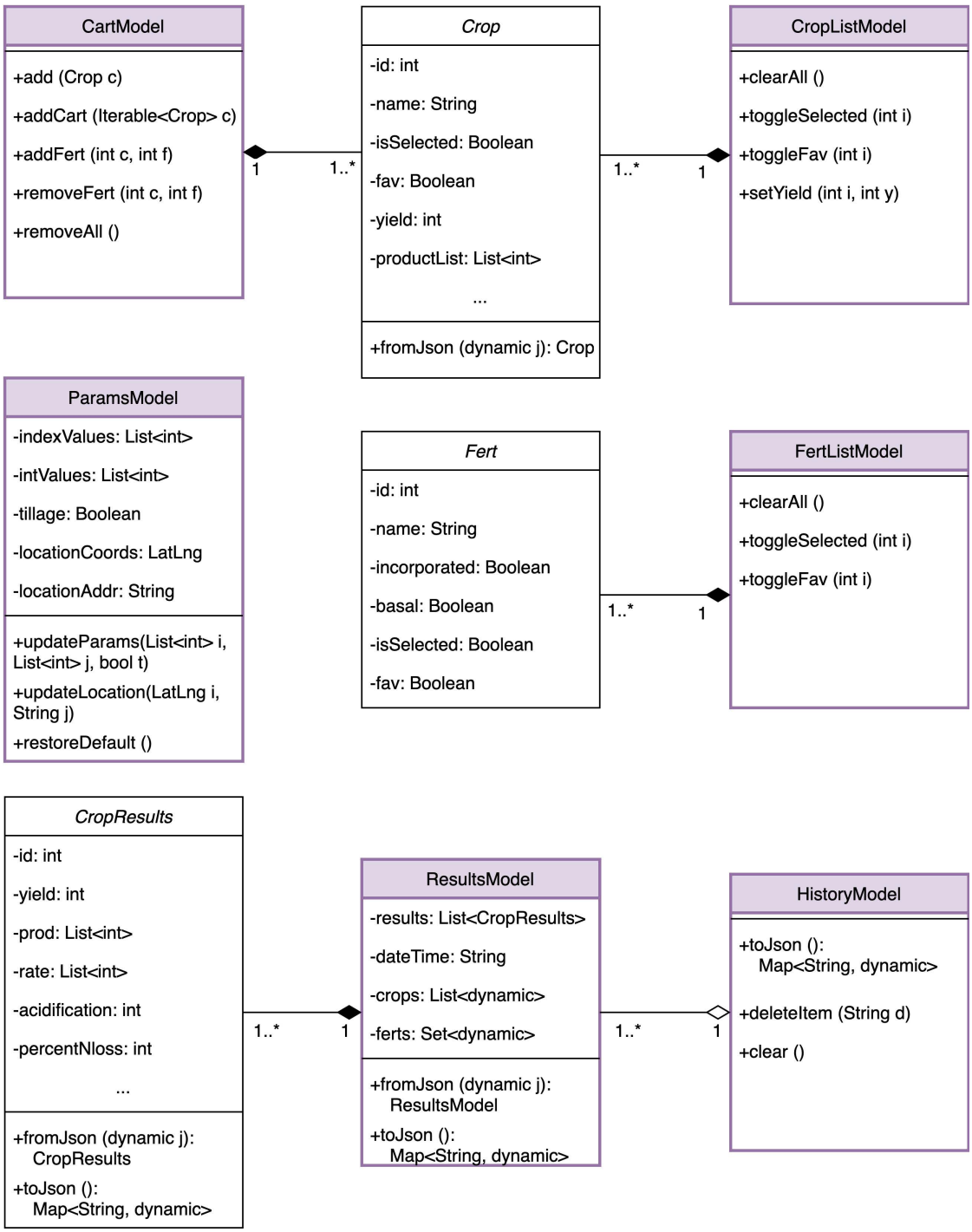


Figure 5-13. Class diagram of the Model module

For the sake of simplicity, we have chosen to keep some basic logic inside these classes. For instance, some of the simple data classes have methods that allow converting the object to JSON or vice versa, and some of the `ChangeNotifier` classes have methods that perform simple operations on their data.

5.3.1 Simple data classes

- *Crop* – Defines some properties, a constructor, and a method to convert directly from JSON to a Crop object. An excerpt of the code is shown below:

```
class Crop {
  final int id;
  String name;
  int yield;
 ...
  bool isSelected = false;
  bool fav = false;
 ...
  List<int> prod = []; //Indexes of assigned fertilizers

  Crop(this.id);

  Crop.fromJson(Map<String, dynamic> json)
      : id = json['id'],
        latinName = json['latin'],
    ...
        kHarv = json['KHARV'].toDouble();
}
```

Crop class

- *Fert* – Analogously to Crop, it defines a Fert object that represents a fertilizer.
- *CropResults* – Encapsulates the results of a calculation for a given crop, including the fertilizers used and the rate for each one. Its `fromJson` and `toJson` methods are vital for storing results; we save results as strings in internal memory, and we use conversion to and from JSON as the intermediate step to do so.

5.3.2 *ChangeNotifier* classes

- *CropListModel* – Encapsulates a list of Crop objects and defines some methods to manipulate the list or a specific Crop in the list. Some of these methods include a call to `NotifyListeners()`, which tells the UI to rebuild and show the new data injected by Provider.
- *FertListModel* – Structured very similarly to the previous class, it holds a list of Fert objects.
- *CartModel* – Encapsulates a list of those Crop objects selected by the user in Crop List Screen. This simplifies the treatment of the data for the chosen crops and greatly reduces the size of the list that is handled by the following screens. It is worth mentioning the `addFert` method, which is called in Fert List Screen to add a specific fertilizer to a crop that was previously chosen for the rotation.

```
import 'package:fertilicalc/models/crop.dart';

class CartModel extends ChangeNotifier {
  List<Crop> items = [];

  ...
  void addFert(int cropIndex, int fertIndex) {
    if (!items[cropIndex].prod.contains(fertIndex)) items[cropIndex].prod.add(fertIndex);
  }
}
```

CartModel class

- *ParamsModel* – Holds data relating to the global parameters and the location chosen by the user. It defines a method to update the values of the global parameters, and another to update the location. These are both called after the user exits the Params Screen.
- *ResultsModel* – Encapsulates the data associated with a single calculation. This includes a list of CropResults objects (one for each crop in the rotation), the date and time when the calculation was made, and some additional fields used for storing and sharing the results.
- *HistoryModel* – Encapsulates a list of *ResultsModel* objects, which correspond to past calculations made by the user.

5.4 Services and Backend

The Services module, following the proposed architecture, interacts with anything outside the application sandbox. In our case, it means interacting with the local storage of the device, and with the Internet services the app relies on. Commands call Services to perform these interactions and potentially update the Model.

The diagram below illustrates the integration of our app's architecture with the backend it communicates with:

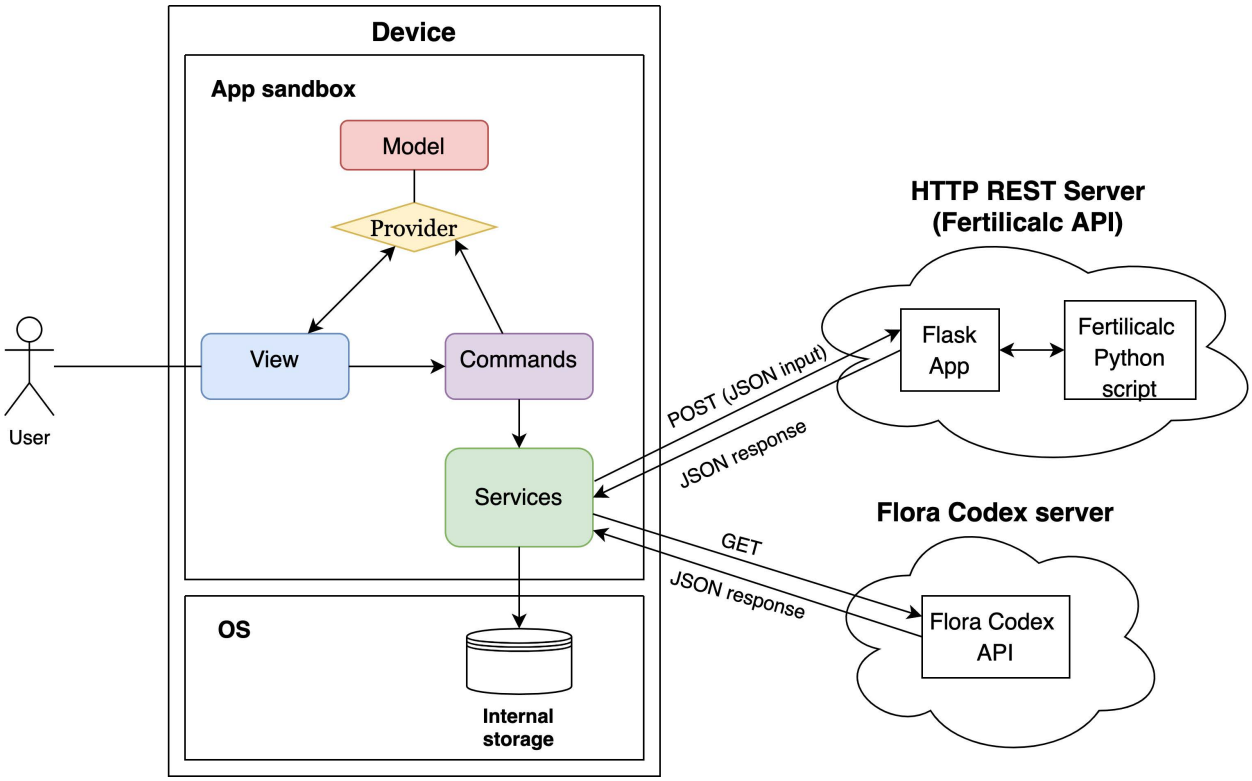


Figure 5-14. Systems diagram of the app and backend

The REST API server which hosts the Fertilicalc Python script that performs the calculations was set up by us (its operation is described in Appendix A). The essential reason for this decoupling is that Flutter cannot run a Python script inside the app sandbox.

The figure below illustrates the process of interaction with our API:

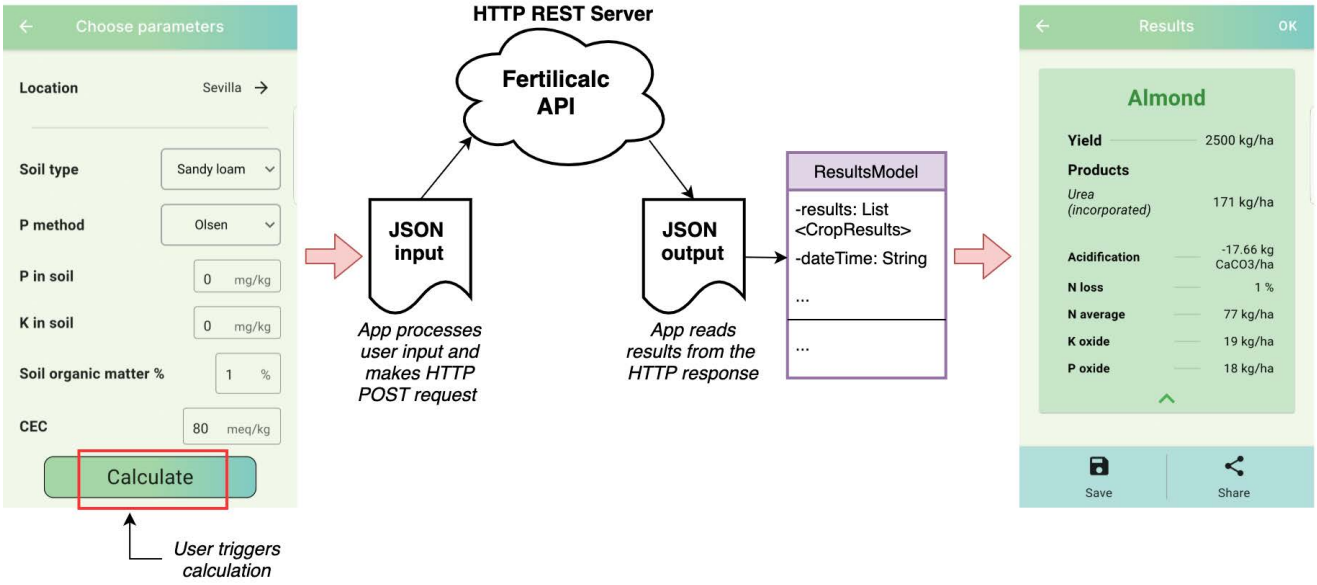


Figure 5-15. Interaction with Fertilicalc API

The Params Screen triggers the calculation of results, then the app sends the user input to our server. The server returns the results, which the app encapsulates inside the Results Model. Lastly, the Results Screen displays this data.

The Flora Codex API is an easily accessible repository of botanical information for an exhaustive collection of plants. [36] We decided to use it to reinforce the app's educational aim. The figure below illustrates the interaction with this API:

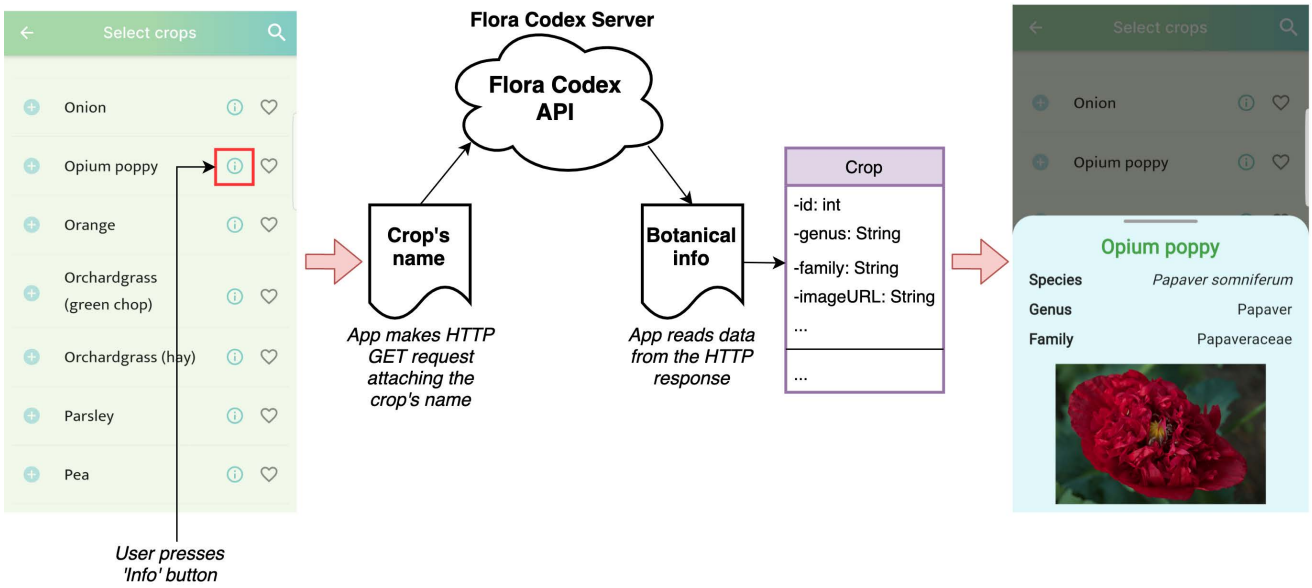


Figure 5-16. Interaction with Flora Codex API

We only need a crop's name to query info about the associated species. We use the returned data to update the appropriate *Crop* object inside the *CropListModel*; then, we display a bottom sheet with the species, genus, family, and a photo of the crop.

Next, we will examine the classes we have defined for the Services module.

5.4.1 Networking

Our app needs to interact with two REST APIs: the Fertilicalc API which runs the Fertilicalc script, and the Flora Codex API.

For this task we have defined the *Networking* service, which offers two asynchronous methods:

- *fetchFromAPI:* makes a POST request to the Fertilicalc server. In its body, it attaches the JSON string which contains the user input. If the call is successful, it returns the response from the server, which contains the results of the calculation.

```
import 'package:http/http.dart' as http;

class Networking {
  //Takes JSON input, makes a request and returns the response (as map)
  Future<String> fetchFromAPI(String jsonInput) async {
    var _url = "https://steamroll24.eu.pythonanywhere.com/output";

    final http.Response res = await http.post(
      Uri.parse(_url),
      headers: <String, String>{
        'Content-Type': 'application/json; charset=UTF-8',
      },
      body: jsonInput,
    );

    if (res.statusCode == 200)
      return res.body;
    else
      throw Exception('Failed to connect to server');
...
```

fetchFromAPI method

- *fetchFromFlora:* makes a GET request to the Flora Codex server, passing as input the name of the crop for which we want to acquire information. If the call is successful, it returns the response from the server, which contains the botanical information for the crop.

It is left to the Commands to parse the responses from the APIs and perform additional filtering of the data returned by the APIs.

5.4.2 Local storage

The *LocalStorage* service interfaces with the *SharedPreferences* plugin. This plugin is used for persisting simple key-value data on disk, in both Android and iOS. It is generally recommended for storing simple data, but not critical data. [37] For more complex data or a larger amount of it, it would be advisable to use a database such as *SQLite*.

```
import 'package:shared_preferences/shared_preferences.dart';

class LocalStorage {
  SharedPreferences _prefs;

  void saveToDisk<T>(String key, T content) async {
    _prefs = await SharedPreferences.getInstance();
    if (content is String)             _prefs.setString(key, content);
    if (content is bool)               _prefs.setBool(key, content);
    if (content is int)                _prefs.setInt(key, content);
    if (content is double)             _prefs.setDouble(key, content);
    if (content is List<String>)       _prefs.setStringList(key, content);

  dynamic getFromDisk(String key) async {
    _prefs = await SharedPreferences.getInstance();
    var value;
    if (_prefs.containsKey(key)) value = _prefs.get(key);
    return value;

  } ...
```

LocalStorage service

We have chosen this approach because of its versatility and ease of use; with a few lines, we can write or read to memory independently of the device's platform. Even though it is limited to simple data types (int, double, string, etc.), it is possible to store more complex objects by encoding nested JSON objects into strings. This is how we store results in history, using the `fromJson` and `toJson` methods mentioned in the previous section.

5.5 Commands

Commands implement business logic that deals with different sets of data, respond to events triggered by the user, and communicate with external services. Essentially, they act as glue between the different layers, along with *Provider*, which binds data directly between the View and the Model.

In this section, we will explore different commands we have defined and how they perform some of the tasks needed to achieve the functionality of the app. These commands all extend a `BaseCommand`, which injects them with the dependencies they need.

5.5.1 BaseCommand

The `BaseCommand` class provides our Commands with the dependencies they need, namely a `BuildContext`[9] and instances of the models supplied by the Provider. The *main.dart* file is in charge of passing it the `BuildContext` it uses to fetch dependencies. This allows us to call the Commands from anywhere in the View, without worrying about references to the Context or access to the Provider.

```
import 'package:provider/provider.dart';
...
BuildContext mainContext;
void init(BuildContext c) => mainContext = c;

class BaseCommand {
  // Models supplied by Provider
  CartModel cartModel = mainContext.read<CartModel>();
  CropListModel cropListModel = mainContext.read<CropListModel>();
  FertListModel fertListModel = mainContext.read<FertListModel>();
  ParamsModel params = mainContext.read<ParamsModel>();
  ResultsModel resultsModel = mainContext.read<ResultsModel>();
  HistoryModel historyModel = mainContext.read<HistoryModel>();
}
```

BaseCommand class

5.5.2 Calculation of new results

For the calculation of new results, we have defined the `LoadResults` command. The Params Screen invokes it when the user presses the 'Calculate' button. Then, while it calculates the results, it navigates to the Results Screen, where the results are displayed (if they were properly loaded).

This command follows the following execution flow:

- It gathers all the user input (list of crops and fertilizers, and parameters), fetching it from the Provider.
- It calls a helper function that, based on the user input, returns the appropriate JSON string to attach to the HTTP request.
- It calls the `fetchFromAPI` service, passing it the JSON input; then it receives its response, which should contain the results for the calculation.

[9] A *BuildContext* is a reference to the location of a specific widget within the tree structure of all the widgets which are built.

- If the service call was successful, the Results Model is updated with the returned data. Another helper function is called to convert the raw response (in JSON) into the appropriate Dart object.

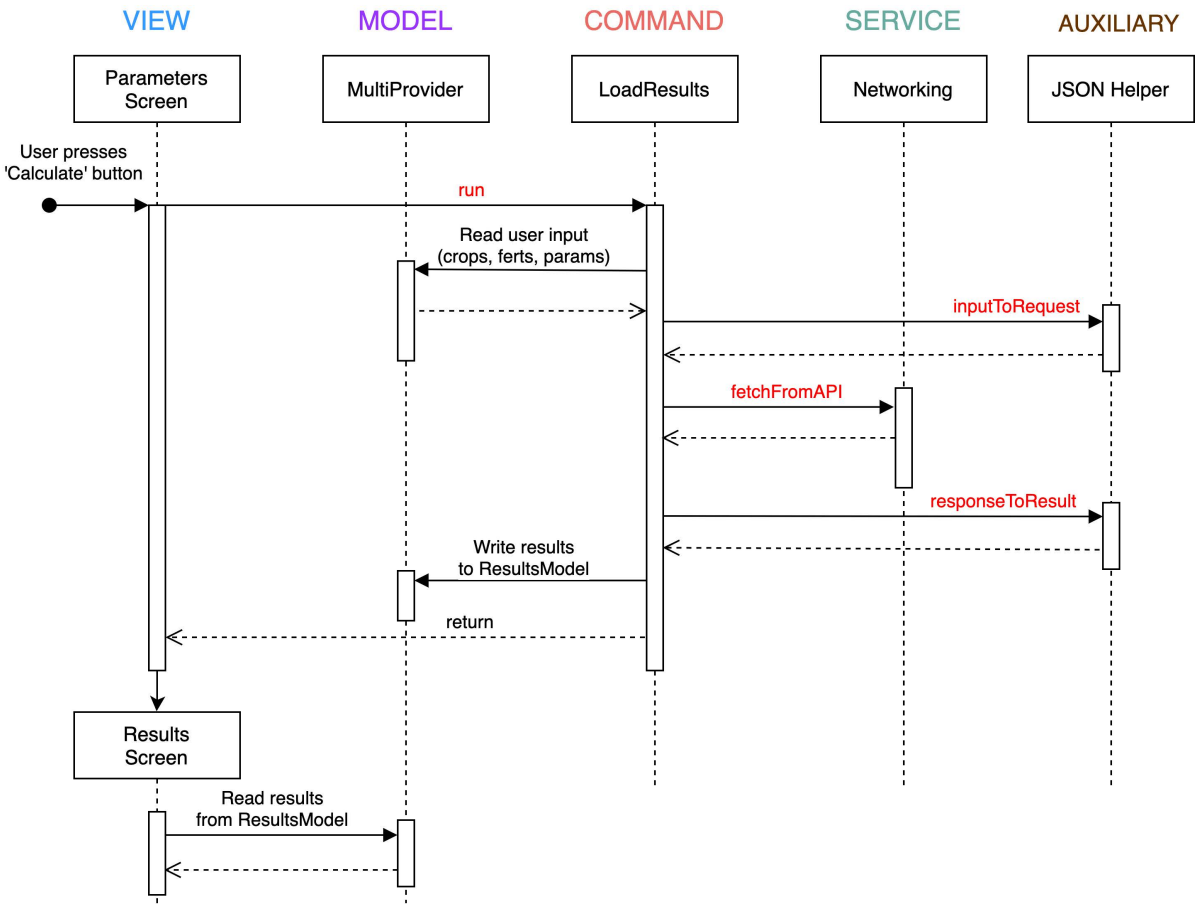


Figure 5-17. Sequence diagram of the calculation of results

Once the results have been loaded into the Provider (by writing them in the Results Model), the user is directed to the Results Screen. This screen reads the results from the Results Model and displays them.

This decoupling of data (Model), logic (Commands and Services), and presentation (View) allows us to reuse the Results Screen for displaying saved results, apart from newly calculated ones. The screen displays the data it reads from the Results Model, independently of where it came from, so we only need to define a specific command to load the results from where we choose, be it internal memory, Internet service, cloud storage, or other.

Note that the Params Screen and the Results Screen are entirely decoupled, since data is not passed directly between them, but rather indirectly through the Provider. This facilitates the rearrangement of the application flow; we could easily, for instance, add another screen where the user checks his selection or tweaks additional parameters before performing the calculation.

5.5.3 History management

We have defined a series of commands to manage the saving and loading of results from history.

To save a result, we use the `saveResult` command. This command first obtains the date and time and stores them in the `ResultsListModel`, which already contains the results we want to save.

It converts this `ResultsListModel` to a JSON object with its `toJson` function, then encodes it into a string. Next, it gets the saved results from internal memory, converts them to a list of strings, and adds the new result to it. Lastly, it saves the updated list of results.

```
Future<void> saveResult() async {
  DateTime now = DateTime.now();
  String dateTime = DateFormat('yyyy-MM-dd – kk:mm').format(now);
  resultsListModel.dateTime = dateTime;

  //We get the JSON corresponding to the current resultsModel
  var _newJson = resultsListModel.toJson();
  var _newString = jsonEncode(_newJson);
  var _history = await LocalStorage().getFromDisk('results') ?? [];
  _history.map((e) => e as String).toList(growable: true);
  _history = <String>[..._history, _newString];
  LocalStorage().saveToDisk('results', _history);
}
```

saveResult command

Below is the sequence diagram illustrating the interactions between the different modules of the app that take place when saving a result:

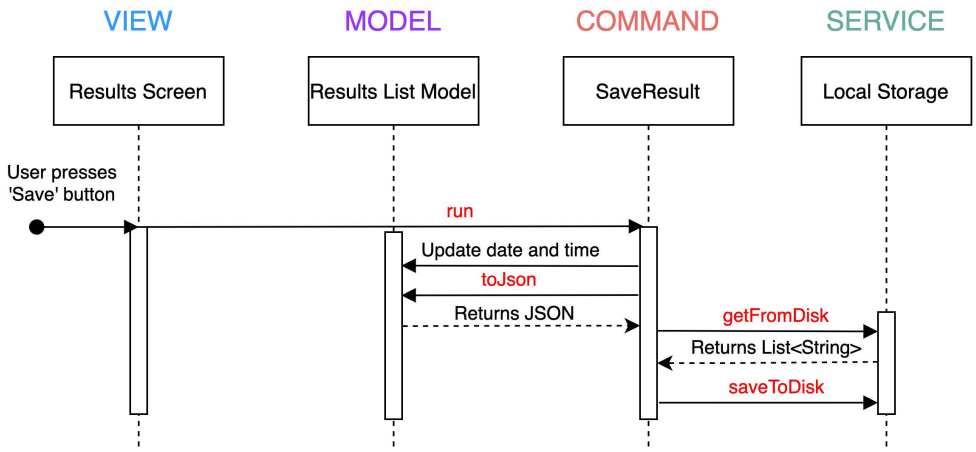


Figure 5-18. Sequence diagram of the storage of results

To allow for the reading of stored results, we have defined the `loadHistory` command. This command loads the entire history from internal memory and stores it in the History Model, so that it is made accessible by Provider and can be shown in the History Screen.

```
Future<void> loadHistory() async {
  historyModel.items = [];

  List<dynamic> _resultsHistory = [];
  _resultsHistory = await LocalStorage().getFromDisk('results');

  //Put results history in Provider's historyModel
  if (_resultsHistory != null) {
    for (var i = 0; i < _resultsHistory.length; i++) {
      var _item = ResultsListModel.fromJson(jsonDecode(_resultsHistory[i]));
      historyModel.items.add(_item);
    }
  }
}
```

loadHistory command

5.5.4 Favorites management

To add a crop to favorites or to remove it, the Crop List Screen interacts directly with the Model through the `toggleFav` method, which changes the value of the `fav` property for the chosen crop.

To save the favorites in memory, we use the `saveCropFavs` command. This command will be called only when the user changes the screen, to minimize the number of memory operations.

```
void saveCropFavs() async {
  //Makes a list with the ID of fav crops
  List<int> favs = [];
  cropListModel.items.where((element) => element.fav == true)
      .forEach((element) => favs.add(element.id));
  //Converts to string list and saves in sharedPrefs
  List<String> strList = favs.map((i) => i.toString()).toList();
  LocalStorage().saveToDisk('cropsFavsList', strList);
}
```

saveCropFavs command

First, it makes a list with the ID of crops that have been marked as favorite. It converts this list to a string list and saves it to memory.

To fully implement R-05, we also define the `loadCropFavs` command, which loads the favorites from memory into the Crop List Model, so that the favorites persist between executions.

```
void loadCropFavs() async {
  List<dynamic> _savedStrList = await LocalStorage().getFromDisk('cropsFavsList') ?? [];

  //Converts to list of ints, updates each fav crop
  if (_savedStrList.isNotEmpty) {
    List<int> _favs = _savedStrList.map((i) => int.parse(i)).toList();
    cropListModel.items.where((element) => _favs.contains(element.id)).forEach((element) {
      element.fav = true;
    });
  }
}
```

loadCropFavs command

First, it reads from memory the string list that contains the favorites. It converts this to a list of integers and updates the crops in Crop List Model so that favorites are marked as such.

The sequence diagram below shows how a favorite is first added, and how the updated list is then saved to memory:

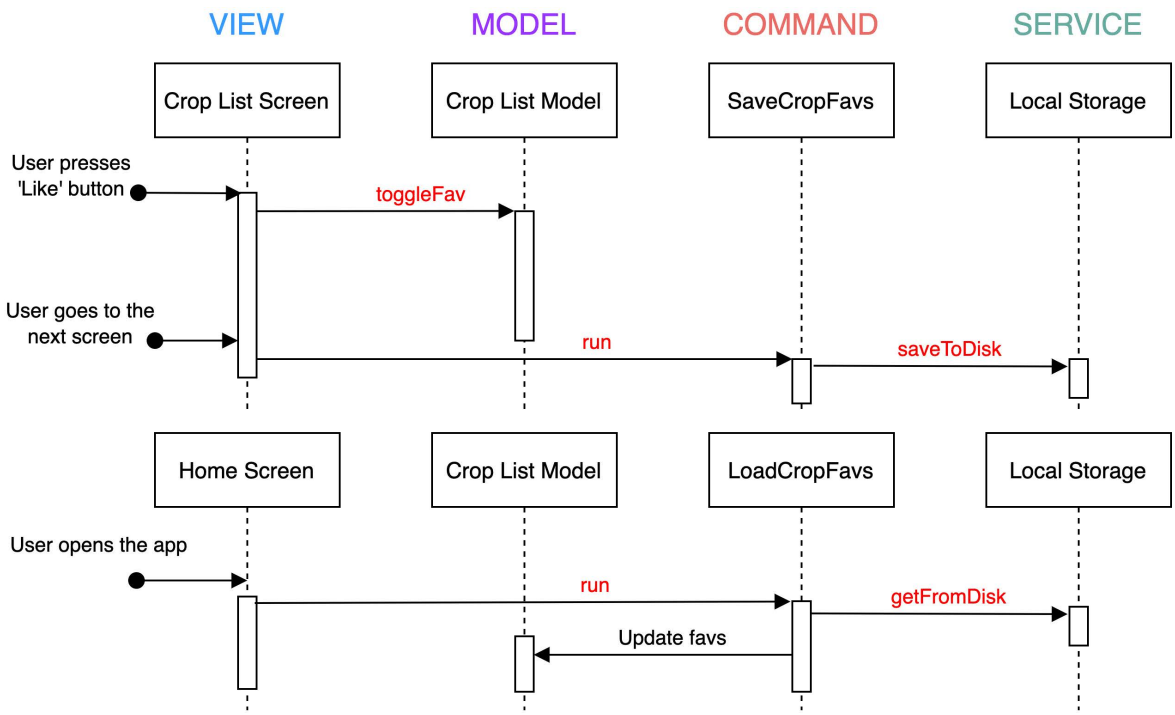


Figure 5-19. Sequence diagram of favorites persistence

For managing fertilizer favorites, we have defined the analogous commands `saveFertFavs` and `loadFertFavs`.

5.5.5 Other commands

We have defined additional commands to perform various other tasks, which we summarize below:

- **loadHistoryItem.** Puts a specific result from history in the Results List Model, so that the Results List Screen can display this set of results.
- **deleteHistory.** Deletes the entire history from local storage.
- **deleteResult.** Retrieves the history from local storage, edits it to remove a specific set of results, and saves it.
- **clearAllData.** Deletes all data stored in local storage: favorites, location, history, etc.
- **shareResult.** Prepares a string that contains the results, which is then shared using the third-party package *share_plus*.
- **botanicalInfoLoad.** Prepares the GET request to send to the Flora Codex API, makes the request through the `Networking` service, and updates the Crop List Model with the info it retrieves.
- **initDataLoad.** Initializes the list of crops and fertilizers, as well as the parameters, which are stored as clear text inside the app files. It also initializes the user location, which it retrieves from local storage.
- **saveLocation.** Saves latitude, longitude, and address in local storage.
- **prepareCart.** Called when navigating from the Crop List Screen to the Fert List Screen, it adds the selected crops to the Cart Model.
- **loadLanguage.** Retrieves the language chosen by the user from local storage.
- **setLanguage.** Sets up the app so that it displays texts in the currently chosen language and saves the chosen language in memory.

Through the implementation and integration of the different modules, we fulfill all the requirements proposed in section 3.2. In Appendix C we include a table outlining the classes from each module involved in delivering each requirement.

5.6 Internationalization

Internationalization consists in writing the app in a way that makes it possible to localize values like texts and layouts for each language that the app supports. Flutter provides widgets and classes that help with internationalization and the Flutter libraries themselves are internationalized. [38]

Reaching the widest user base possible is an important aspiration of this project, so we have decided to include 7 languages: English, Spanish, Italian, Russian, Chinese, Hindi, and Arabic. The Fertilicalc desktop app has versions in 88 languages [4]. In many of these versions, translations to foreign languages were made in collaboration with agriculture specialists from abroad. [5] We have incorporated these translations[10] into our app through local JSON files and use them to display localized texts throughout the screens.

There is one JSON file for each supported language. Below is an excerpt of one of these JSON files, which contains the translations for Chinese (*zh.json*):

```
{
  "crops": {
    "1": "苜蓿 (鲜切/营养)",
    "2": "苜蓿 (鲜切/营养)",
    "3": "苜蓿（干草/营养）",
    …
  },
  "ferts": {
    "1": "硝酸钠",
    "2": "硝酸钙",
    "3": "硝酸镁",
    …
  },
  "technical": {
    "Yield": "产量",
    "kg/ha": "公斤/公顷",
    "Soil data": "土壤数据",
    …
  },
  "layout": {
    "Select crops": "選擇農作物",
    "Select fertilizers": "選擇肥料",
    "Choose parameters": "選擇參數",
    …
  }
}
```

JSON – Language file for Chinese

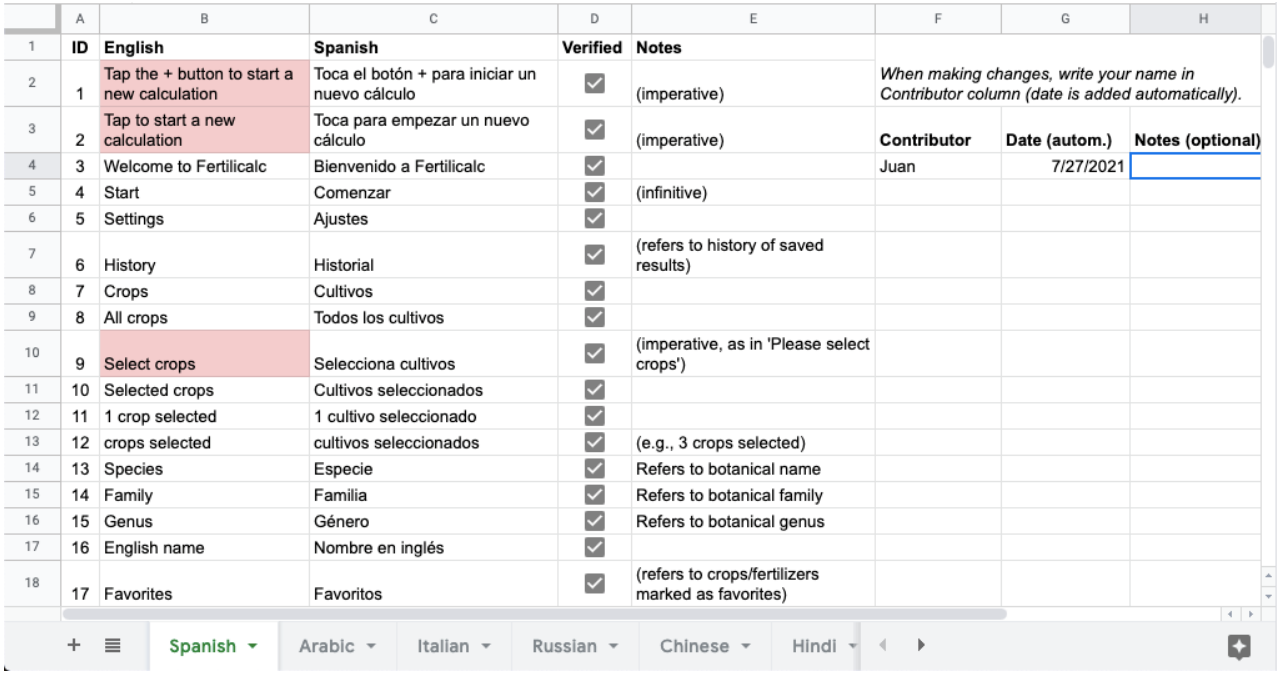
The incorporated translations are divided into four fields: '*crops*', for the names of crops; '*ferts*', for the names of fertilizers; '*technical*', for names of parameters, units, and various technical terms shown throughout the app; and '*layout*', for texts which are specific to the app's screens.

The first three fields comprise the translations that we adopted from the desktop app, while '*layout*' includes terms that were not included in those translations. We mostly used Google Translate to perform these additional translations and the Microsoft Terminology Collection for select terms.

To refine the quality of the '*layout*' translations, we have created a collaborative Google Sheets page. This

[10] The translations were provided courtesy of the author of the Fertilicalc desktop software.

spreadsheet has a sheet for each of the languages supported by the app and can be edited by invited users. It is meant to facilitate the correction of various translations and could be shared with native users who wish to collaborate in this task.



	A	B	C	D	E	F	G	H
1	**ID**	**English**	**Spanish**	**Verified**	**Notes**			
2	1	Tap the + button to start a new calculation	Toca el botón + para iniciar un nuevo cálculo	☑	(imperative)	*When making changes, write your name in Contributor column (date is added automatically).*		
3	2	Tap to start a new calculation	Toca para empezar un nuevo cálculo	☑	(imperative)	**Contributor**	**Date (autom.)**	**Notes (optional)**
4	3	Welcome to Fertilicalc	Bienvenido a Fertilicalc	☑		Juan	7/27/2021	
5	4	Start	Comenzar	☑	(infinitive)			
6	5	Settings	Ajustes	☑				
7	6	History	Historial	☑	(refers to history of saved results)			
8	7	Crops	Cultivos	☑				
9	8	All crops	Todos los cultivos	☑				
10	9	Select crops	Selecciona cultivos	☑	(imperative, as in 'Please select crops')			
11	10	Selected crops	Cultivos seleccionados	☑				
12	11	1 crop selected	1 cultivo seleccionado	☑				
13	12	crops selected	cultivos seleccionados	☑	(e.g., 3 crops selected)			
14	13	Species	Especie	☑	Refers to botanical name			
15	14	Family	Familia	☑	Refers to botanical family			
16	15	Genus	Género	☑	Refers to botanical genus			
17	16	English name	Nombre en inglés	☑				
18	17	Favorites	Favoritos	☑	(refers to crops/fertilizers marked as favorites)			

Spanish | Arabic | Italian | Russian | Chinese | Hindi

Figure 5-20. Spreadsheet for correction of translations

To facilitate editing, the ‘English’ and ‘Notes’ columns, which serve as guidance, are automatically replicated across all sheets using the `ArrayFormula` function in Google Sheets.

To automate the process of updating the translations in the app, we have written a Python script that reads the corrected translations from the spreadsheet and updates the app’s JSON language files accordingly:

```
SCOPES = ['https://www.googleapis.com/auth/spreadsheets.readonly']
SPREADSHEET_ID = 'ndE0ROP...'
languages={'Spanish':'es','Arabic':'ar','Hindi':'hi','Italian':'it','Russian':'ru','Chinese':'zh'}

def main():
    #Access the Sheets API
    creds = None
    # ... (ommited code from Google's Python quickstart)
    service = build('sheets', 'v4', credentials=creds)
    sheet = service.spreadsheets()

    #Iterate for each language
    for lang in languages:
        RANGE_NAME = lang+'!B:C';

        # Call the Sheets API
        langSheet = sheet.values().get(spreadsheetId=SPREADSHEET_ID,range=RANGE_NAME).execute()
        values = langSheet.get('values', [])

        #Make a dictionary for the given language
        langDict = {}
        for row in values:
            if(values.index(row)!=0):  # Exclude first row
                langDict.update({row[0]:row[1]})

        #Open language file and load its content in JSON object
        filename = languages[lang]+'.json'
        a_file = open(filename, "r")
        json_object = json.load(a_file)
```

```
        a_file.close()
        #Update the 'layout' field with our dictionary
        json_object['layout'].update(langDict)
        #Write updated object to JSON file
        a_file = open(filename, "w")
        json.dump(json_object, a_file, ensure_ascii=False)
        a_file.close()
```

Python – Script for automating translation updates

The approach we use for localizing texts throughout the app is based on Flutter's guide on internalization (also called 'i18n') [38], and the tutorial found at [39].

This method uses Flutter's *flutter_localizations* library for adapting widgets to the localized texts, and the correct left-to-right and right-to-left layout. When the device's locale is changed, through the device's settings, the widgets throughout the app are adapted to this locale (if it is supported).

For the code to incorporate localized strings, it only needs to call `AppLocalizations.of(context)`. We have customized the `AppLocalizations` class with different methods for the different categories of translated terms. We call `AppLocalizations.of(context).layoutValue(String text)` to obtain a localized text from the "*layout*" section, while we call `AppLocalizations.of(context).cropName(int index)` to obtain the translated name of a crop (see code from section 5.2.2).

To allow a user to override the device's locale and choose a particular language (R-28), we have implemented custom methods (described in section 5.5.5) that are called in the Languages Screen.

Below we show different screens of the app being run in different languages. From left-to-right and top-to-bottom, they show the app being run in Spanish, Italian, Russian, Hindi, Chinese, and Arabic:

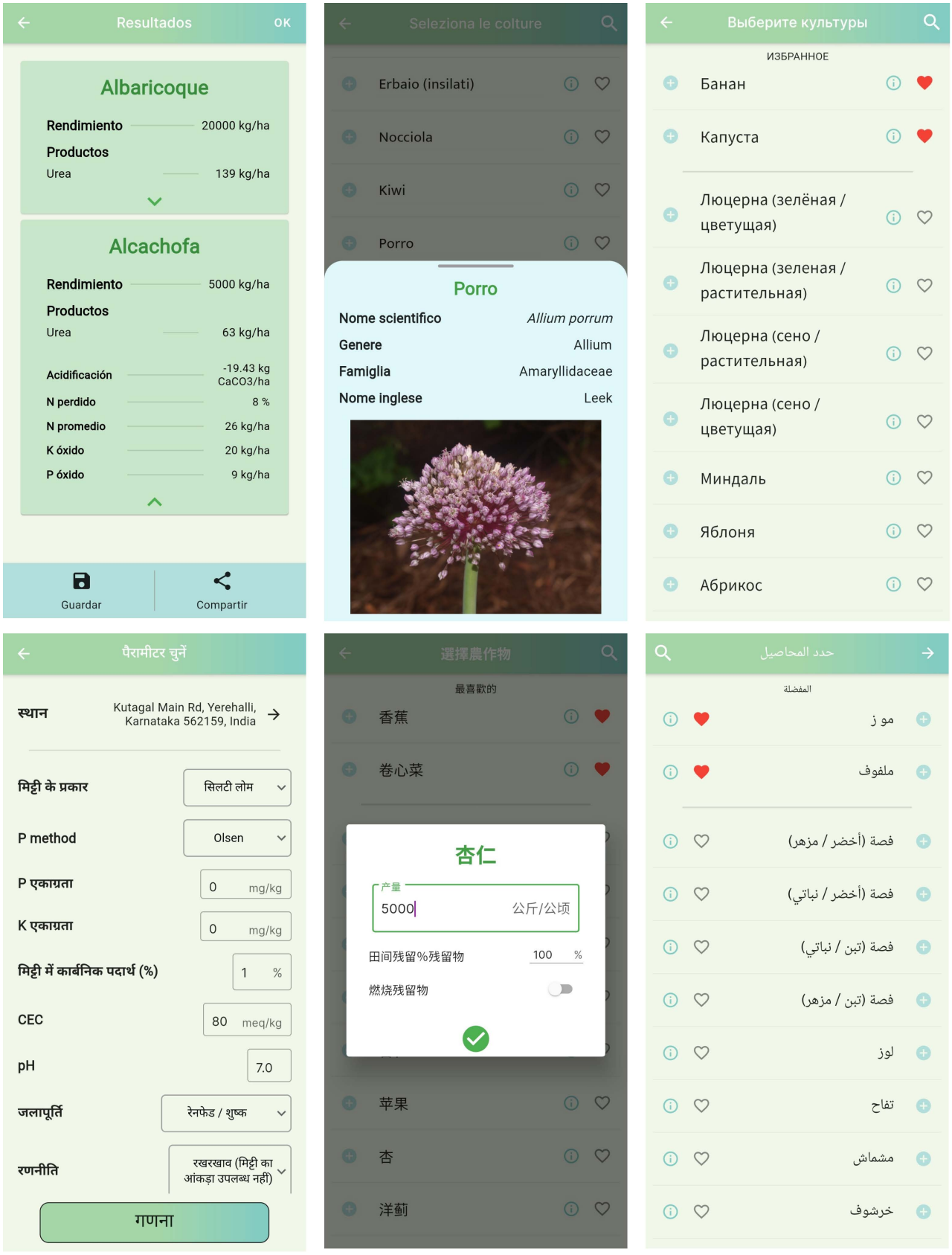


Figure 5-21. Localized screens

Since the Flutter libraries are already internationalized, when the app is executed in Arabic, the Flutter widgets automatically adopt a right-to-left layout.

6 CONCLUSIONS AND FUTURE WORK

As of August 2021, I am in the process of testing a *pre-release* version of the Android app with select testers, through the 'Closed testing' tool offered by the Google Play Console. [40] So far, the testing has shown our app is stable and can be run on various devices.

This chapter offers some conclusions about the project development and its results, as well as some proposals for future work.

6.1 Conclusions

Developing this project was a big challenge; thankfully, the plethora of available documentation and resources for the development of Flutter apps was incredibly helpful.

When I started making the prototype, I did not have any prior knowledge of Flutter and had not programmed a mobile app. However, through the iterative design process, I was able to gradually fill gaps in my knowledge and carry out the development.

While I estimate that the learning curve would have been similar for either Android or iOS, I consider that this project validates the use of Flutter for small or medium-sized apps. Flutter seems to have a promising future, with its ever-growing community of passionate developers and extensive support from the Google team.

Provider has proven to be a solid choice for state management. It was created by a developer but is now maintained in collaboration with the Google team and officially recommended as the state management approach for beginners.

The design process was aided substantially by introducing a structured architecture. Refactoring the code to accommodate this was not particularly labor-intensive, primarily because we maintained our state management approach.

The functional requirements that were established have all been fulfilled. The use of third-party packages was immensely helpful for the implementation of some of these requirements.

I consider the non-functional requirements to be fulfilled as well:

- Size: The size of the current executable of the app is 16.8 MB, below the 20 MB we proposed.
- Usability: The feedback we have received so far has been satisfactory in this respect.
- Internalization: The app has been successfully adapted to the 7 languages we proposed.
- Compatibility: The app can be run on the proposed devices.

I consider the essential aim of the project, which was to develop *Fertilicalc Mobile* as a functional and extensible mobile app, has been accomplished. Moreover, the educational component of the app has been reinforced by incorporating botanical data from the Flora Codex API.

6.2 Future work

There are plentiful improvements that could be made to the app. Additional features are already being worked on, which, due to time constraints, are not included in the version presented in this document.

The following are areas where we consider work should be prioritized to improve the project:

- **Richer set of results.** There is additional data provided by the Fertilicalc script that could be included in the results, such as the global balance of nutrients of the entire rotation. It would be of great interest to include this data in the Results Screen.
- **Pricing of fertilizers.** It could be of interest to provide approximate prices of fertilizers and include an approximate cost for the proposed rates in the results. Obtaining prices on a global scale is not practical; on the other hand, connecting to local fertilizer dealers could be a viable option. Adding advertising from these local dealers could also serve as a potential way of sponsoring the app. Another simpler but less accurate option would be to use a *web scraping* script that aggregates fertilizer prices from *Alibaba*.
- **Cloud integration.** The addition of a sign-in screen and integration with cloud storage would allow users to access their saved data on any device they choose. Users could create an account or use an external service (e.g., Google or Apple) to sign in. It would be of special interest to integrate the app with Firebase backend services, due to the extensive support and documentation already available. [41]
- **Testability.** Due to the low complexity of our app and time constraints, we have only performed manual tests. The inclusion of automated testing would be tremendously beneficial for its long-term maintainability and scalability. It would also be advantageous to use *Continuous Integration* services, which allow running tests automatically when pushing new code changes. [42]
- **Code maintenance.** Flutter and Dart are still relatively young technologies (Flutter 1.0 was released in December 2018). Since the app's development started, there have been significant updates in the framework, and in some of the third-party packages we have incorporated. As a result, it is imperative in the short to medium term to accommodate these updates in our code. One major pending tasks regarding this maintenance is updating the code to Flutter 2, which includes migrating to null safety. [43]
- **Usability.** An important improvement for ease-of-use would be to include informative prompts at different points so that users can have a better understanding of some of the technical terms and functionality involved. Another valuable option would be to include a brief in-app tutorial that provides a walkthrough of the app. We also propose measuring usability more thoroughly, e.g., through a feedback form.
- **Responsiveness.** A *responsive* app is one that has had its layout tuned for the available screen size. Responsive design ensures that a wider range of users can have a seamless experience, no matter the size of their device. Flutter widgets already provide a lot of responsiveness themselves. [44] However, our code should be updated to improve this; for instance, the font size of texts should be adapted to the device's screen size.
- **Internationalization.** The translation of crops, fertilizers, and other technical terms has already been made by native contributors for many other languages. [5] Future versions of the app should include more of these languages, starting with the most widely spoken ones, such as Portuguese or Persian. Additional work should be done to ensure the quality of the Right-To-Left layouts. Also, foreign collaborators should be invited to participate in the correction of translations, making use of the mechanism we proposed in section 5.6.
- **iOS support.** Even though the execution of the app on iOS is successful, there is still work left to do to assure the same quality and stability as on Android. Testing still needs to be performed on physical devices. Additionally, some adjustments could be made to the widgets and layout to further tailor the UI to suit iOS design when the app is executed on iOS.

APPENDIX A: INSTALLATION OF REST API

To perform the calculations that take the user's input and return the appropriate insight, similarly to the desktop *Fertilicalc* desktop app, we have employed a Python script. This script was provided courtesy of the desktop version's author; a version of it is available to download at [4]

The original script takes its input as a *.txt* file and exports its output as a *.csv* file. To facilitate the communication between our app and the script, we have modified the script so that the input and output are JSON objects.

Since a Flutter app cannot run a Python script itself, we have deployed a web server based on the micro web framework Flask. This server exposes a REST API; it takes requests from the app, runs the script, and returns responses that contain the results.

We have hosted the web server, along with the script, at *PythonAnywhere*, an online integrated development environment and web hosting service based on Python. [10]

The following is the code that runs our web server:

```
from flask import Flask
from flask import request
from flask import jsonify
import script
import json

app = Flask(__name__)

@app.route('/output', methods = ['POST'])
def demo():
    content = request.get_json()
    return jsonify(script.fun(content))
```

Python – *Flask* web server

The **/output** endpoint of our web server takes the JSON input from the POST request (`content`), passes it to the `script.fun` function and returns the script's output as JSON. The `jsonify` function is used to convert a Python dictionary to JSON.

The following is an example of an input that our app sends to the web server and is processed by the script:

```
{
    "params": {
        "soil_n": 2,
        "p_conc": 1,
        "k_conc": 0,
        "ph": 0,
        "p_method": 0,
        "water_supply": 1,
        "strategy": 1,
        "tillage": 0
    },
```

```
    "list": [
        {
            "crop_n": 7,
            "y": 2000,
            "cv": 0,
            "h_i": 60,
            "fres": 100,
            "n_harv": 1,
            "p_harv": 0.14,
            "k_harv": 2,
            "burning": 0
        }
    ],
    "fert": [
        {
            "crop_n": 1,
            "prod_index": 5,
            "code_inc": 1,
            "code_bas": 0
        }
    ],
    "coord": {
        "lat": 37.85,
        "long": -4.8
    }
}
```


JSON – *Fertilicalc* script input

The `params` field contains some global parameters that the user can customize, such as soil type (`soil_n`) or strategy. The `list` field contains a list of the chosen crops, each with its corresponding parameters: an identifier (`crop_n`), yield (`y`), whether residues are burnt (`burning`), etc. The `fert` field contains a list of the chosen fertilizer–crop pairings, each with two additional parameters: `code_inc` and `code_bas`. These correspond to the '*incorporated*' and '*before planting*' parameters from the Fert List Screen, respectively. The `coord` field contains the coordinates of the location chosen by the user.

After performing the necessary calculations, the script returns an output like this:


```
{
    "1": {
        "acidification": 2.622147818136536,
        "crop_n": 7,
        "ffix": 0.0,
        "k_fert": 257.40000000000003,
        "k_fert_oxide": 310.0897800000001,
        "n_fert_avg": 5.133333333333335,
        "n_kg_crop": 7.28,
        "nmin": 0.9333333333333335,
        "p_fert": 42.120000000000005,
        "p_fert_oxide": 96.4548,
        "percent_N_loss": 13.935210770313686,
        "product_1": 5,
        "product_2": 0,
        "product_3": 0,
        "product_4": 0,
        "rate_1": 12.966301770719793,
        "rate_2": 0.0,
        "rate_3": 0.0,
        "rate_4": 0.0
    }
}
```


JSON – *Fertilicalc* script output

It is a JSON list with one element for each crop in the rotation. For each element, it provides various values, like nutrient requirements (`k_fert`, `n_fert_avg`, …), the chosen fertilizer products (`prod_1`, `prod_2`, ...), and their rates (`rate_1`, `rate_2`, ...). This comprises the data that our app displays in the Results Screen.

APPENDIX B: DESIGN EVOLUTION

To illustrate the iterative design and development of the UI, we include here a comparison of three of the app's screens in different stages of the design: the initial screen mockups, the initial prototype, and the final design.

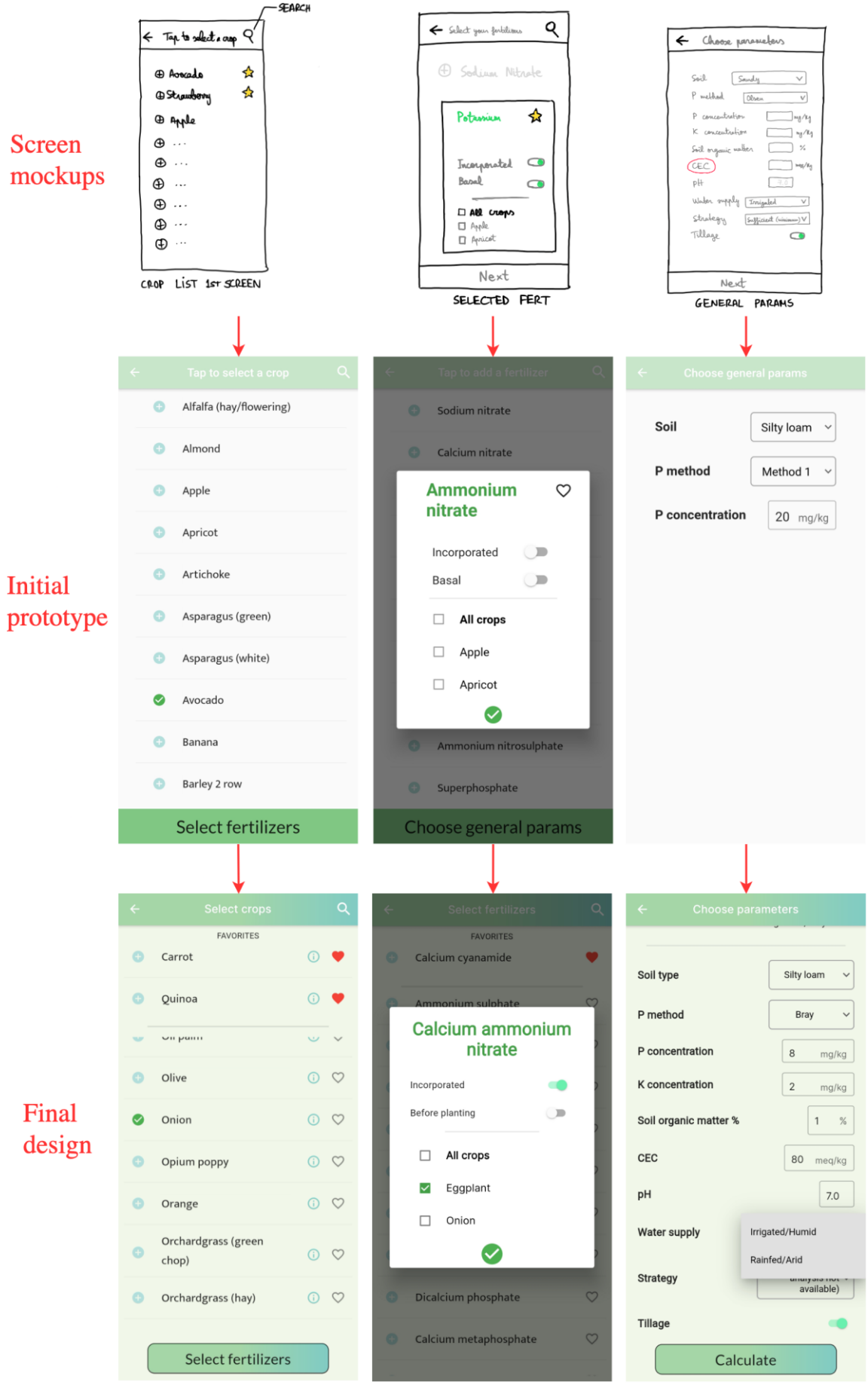


Figure B-1. Timeline of UI design

APPENDIX C: INTEGRATION OF MODULES

The following is the table outlining the classes of each module involved in the fulfillment of each requirement:

ID	View	Commands	Services	Model
R-01	Crop List Screen	initDataLoad		Crop List Model
R-02	Crop List Screen			Crop List Model
R-03	Crop List Screen			Crop List Model
R-04	Crop List Screen			Crop List Model
R-05	Crop List Screen	loadCropFavs, saveCropFavs	LocalStorage	Crop List Model
R-06	Crop List Screen	botanicalInfoLoad	Networking	Crop List Model
R-07	Fert List Screen	initDataLoad		Fert List Model
R-08	Crop List Screen	prepareCart		Crop List Model, Cart Model
R-09	Fert List Screen			Fert List Model
R-10	Fert List Screen			Fert List Model, Cart Model
R-11	Fert List Screen			Fert List Model
R-12	Fert List Screen	loadFertFavs, saveFertFavs	LocalStorage	Fert List Model
R-13	Params Screen, Location Screen			Params Model
R-14	Params Screen, Location Screen	saveLocation, getLat, getLong, getStrAddr, initDataLoad	LocalStorage	Params Model
R-15	Params Screen	initDataLoad		Params Model

R-16	Params Screen			Params Model
R-17	Settings Screen			Params Model
R-18	Params Screen	loadResults	Networking	Cart Model, Params Model
R-19	Results Screen			Results List Model
R-20	Results Screen			Results List Model
R-21	Results Screen	saveResult	LocalStorage	Results List Model
R-22	History Screen	loadHistory		History Model
R-23	History Screen, Results Screen	loadHistoryItem		Results List Model
R-24	Results Screen	shareResult		Results List Model
R-25	Settings Screen	deleteResult	LocalStorage	
R-26	Settings Screen	deleteHistory	LocalStorage	
R-27	Settings Screen	clearAllData	LocalStorage	
R-28	Settings Screen, Languages Screen	loadLanguage, getLanguage		

Figure C-1. Table of integration of modules

REFERENCES

[1] Nifa.usda.gov. 2016. The Nutrient Challenge of Sustainable Fertilizer Management | National Institute of Food and Agriculture. [online] Available at: <https://nifa.usda.gov/blog/nutrient-challenge-sustainable-fertilizer-management> [Accessed 6 June 2021].

[2] Silva, G., 2013. Challenges to feeding the seven billion and beyond – Part 3: Role of fertilizers. [online] MSU Extension. Available at: <https://www.canr.msu.edu/news/challenges_to_feeding_the_seven_billion_and_beyond_part_iii_role_of_fertili> [Accessed 6 June 2021].

[3] Villalobos, F., Delgado, A., López-Bernal, Á. and Quemada, M., 2020. FertiliCalc: A Decision Support System for Fertilizer Management. International Journal of Plant Production, 14(2), pp.299-308.

[4] Villalobos, F., 2021. Principles of Agronomy for Sustainable Agriculture / Fitotecnia General. [online] Uco.es. Available at: <http://www.uco.es/fitotecnia/fertilicalc.html> [Accessed 13 July 2021].

[5] López Bernal, Á., Testi, L., Orgaz, F., Delgado, A., Quemada, M. and Villalobos, F., 2020. Una aplicación windows de apoyo a la docencia del cálculo de las necesidades de fertilizantes. Revista de Innovación y Buenas Prácticas Docentes, 9(1), pp.71-79.

[6] DataReportal – Global Digital Insights. 2021. Digital Around the World — DataReportal – Global Digital Insights. [online] Available at: <https://datareportal.com/global-digital-overview> [Accessed 5 July 2021].

[7] Pew Research Center's Global Attitudes Project. 2021. Emerging Nations Embrace Internet, Mobile Technology. [online] Available at: <https://www.pewresearch.org/global/2014/02/13/emerging-nations-embrace-internet-mobile-technology/> [Accessed 5 July 2021].

[8] Kanza, P. and Vitale, J., 2015. Agriculture in Developing Countries and the Role of Government: Economic Perspectives. [online] Ideas.repec.org. Available at: <https://ideas.repec.org/p/ags/aaea15/205362.html> [Accessed 5 July 2021].

[9] Diagrams.net. 2021. Diagram Software and Flowchart Maker. [online] Available at: <https://www.diagrams.net/> [Accessed 13 July 2021].

[10] LLP, P., 2021. Host, run, and code Python in the cloud: PythonAnywhere. [online] Pythonanywhere.com. Available at: <https://www.pythonanywhere.com/> [Accessed 13 July 2021].

[11] Uptech.team. 2021. Native vs Cross-Platform Development: Pros & Cons Revealed. [online] Available at: <https://uptech.team/blog/native-vs-cross-platform-app-development> [Accessed 5 September 2021].

[12] Statista. 2021. Mobile OS market share 2021 | Statista. [online] Available at: <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/> [Accessed 5 September 2021].

[13] Jamalova, M. and Milán, C., 2019. The Comparative Study of the Relationship Between Smartphone Choice and Socio-Economic Indicators. International Journal of Marketing Studies, 11(3), p.11.

[14] Statista. 2021. Cross-platform mobile frameworks used by global developers 2021 | Statista. [online] Available at: <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/> [Accessed 6 September 2021].

[15] [online] Available at: <https://nix-united.com/blog/flutter-vs-react-native/> [Accessed 5 September 2021].

[16] [online] Available at: <https://trends.google.com/trends/explore?q=flutter,react%20native> [Accessed 6 September 2021].

[17] Medium. 2021. Flutter vs Native vs React-Native: Examining performance. [online] Available at: <https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980> [Accessed 6 September 2021].

[18] Flutter.dev. 2021. Flutter architectural overview. [online] Available at: <https://flutter.dev/docs/resources/architectural-overview> [Accessed 16 June 2021].

[19] Flutter.dev. 2021. Layouts in Flutter. [online] Available at: <https://flutter.dev/docs/development/ui/layout> [Accessed 13 July 2021].

[20] Flutter.dev. 2021. Write your first Flutter app, part 1. [online] Available at: <https://flutter.dev/docs/get-started/codelab> [Accessed 6 September 2021].

[21] Api.flutter.dev. 2021. AppBar class - material library - Dart API. [online] Available at: <https://api.flutter.dev/flutter/material/AppBar-class.html> [Accessed 6 September 2021].

[22] Appinventiv. 2020. Explained: Mobile App Architecture - The Basis of App Ecosystem. [online] Available at: <https://appinventiv.com/blog/mobile-app-architecture-explained/> [Accessed 13 July 2021].

[23] Flutter.dev. 2021. Simple app state management. [online] Available at: <https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple> [Accessed 13 July 2021].

[24] Faust, Sebastian. 2021. “Using Google´s Flutter Framework for the Development of a Large-Scale Reference Application.” | Available at: <https://epb.bibl.th-koeln.de/frontdoor/deliver/index/docId/1498/file/flutter-for-the-dev-of-large-scale-ref-app.pdf> [Accessed 5 September 2021]

[25] raywenderlich.com. 2021. State Management With Provider. [online] Available at: <https://www.raywenderlich.com/6373413-state-management-with-provider> [Accessed 13 July 2021].

[26] Dart packages. 2021. provider | Flutter Package. [online] Available at: <https://pub.dev/packages/provider> [Accessed 13 July 2021].

[27] Magora Systems. 2018. Overview of mobile app development architecture. [online] Available at: <https://magora-systems.com/mobile-app-development-architecture/> [Accessed 13 July 2021].

[28] en.wikipedia.org. 2021. Multitier architecture - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Multitier_architecture#Three-tier_architecture> [Accessed 13 July 2021].

[29] gskinner blog. 2020. Flutter: State Management using an MVC+S Architecture - gskinner blog. [online] Available at: <https://blog.gskinner.com/archives/2020/09/flutter-state-management-with-mvcs.html> [Accessed 13 July 2021].

[30] Dart packages. 2021. convex_bottom_bar | Flutter Package. [online] Available at: <https://pub.dev/packages/convex_bottom_bar> [Accessed 13 July 2021].

[31] Api.flutter.dev. 2021. Navigator class - widgets library - Dart API. [online] Available at: <https://api.flutter.dev/flutter/widgets/Navigator-class.html> [Accessed 13 July 2021].

[31] Api.flutter.dev. 2021. ListView.separated constructor - ListView class - widgets library - Dart API. [online] Available at: <https://api.flutter.dev/flutter/widgets/ListView/ListView.separated.html> [Accessed 13 July 2021].

[32] Dart packages. 2021. permission_handler | Flutter Package. [online] Available at: <https://pub.dev/packages/permission_handler> [Accessed 13 July 2021].

[33] Dart packages. 2021. google_maps_place_picker | Flutter Package. [online] Available at: <https://pub.dev/packages/google_maps_place_picker> [Accessed 13 July 2021].

[34] Dart packages. 2021. settings_ui | Flutter Package. [online] Available at: <https://pub.dev/packages/settings_ui> [Accessed 13 July 2021].

[35] Dart packages. 2021. url_launcher | Flutter Package. [online] Available at: <https://pub.dev/packages/url_launcher> [Accessed 13 July 2021].

[36] Floracodex.com. 2021. Flora Codex. [online] Available at: <https://floracodex.com/> [Accessed 13 July 2021].

[37] Dart packages. 2021. shared_preferences | Flutter Package. [online] Available at: <https://pub.dev/packages/shared_preferences> [Accessed 13 July 2021].

[38] Flutter.dev. 2021. Internationalizing Flutter apps. [online] Available at: <https://flutter.dev/docs/development/accessibility-and-localization/internationalization> [Accessed 6 September 2021].

[39] Reso Coder. 2021. Flutter Localization the Easy Way – Internationalization with JSON - Reso Coder. [online] Available at: <https://resocoder.com/2019/06/01/flutter-localization-the-easy-way-internationalization-with-json/> [Accessed 13 July 2021].

[40] Play.google.com. 2021. Closed testing | Google Play Console. [online] Available at: <https://play.google.com/console/about/closed-testing/> [Accessed 13 July 2021].

[41] Flutter.dev. 2021. Firebase. [online] Available at: <https://flutter.dev/docs/development/data-and-backend/firebase> [Accessed 13 July 2021].

[42] Flutter.dev. 2021. Testing Flutter apps. [online] Available at: <https://flutter.dev/docs/testing> [Accessed 13 July 2021].

[43] Flutter.dev. 2021. Null safety in Flutter. [online] Available at: <https://flutter.dev/docs/null-safety> [Accessed 13 July 2021].

[44] Flutter.dev. 2021. Creating responsive and adaptive apps. [online] Available at: <https://flutter.dev/docs/development/ui/layout/adaptive-responsive> [Accessed 8 September 2021].

[45] Flutter.dev. 2021. Flutter documentation. [online] Available at: <https://flutter.dev/docs> [Accessed 13 July 2021].

[46] https://gitfront.io/r/user-2169473/30781c2550f4913e07d246452bed597907fe4f8f/fertilicalcMobile/

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Fertilicalc Mobile: Aplicación en Flutter para gestión de fertilizantes

(Resumen en castellano)

Autor:
Juan Villalobos Carrasco

Tutor:
Juan Manuel Vozmediano Torres
Profesor titular

Dpto. de Ingeniería de Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Índice

1 INTRODUCCIÓN

Uno de los retos más importantes para la humanidad es la producción de suficiente comida para alimentar a una población creciente. Se estima que la población global llegará a 9,7 mil millones para 2050, y la producción agrícola tendrá que duplicarse para entonces.

En países en desarrollo, especialmente, la disponibilidad y asequibilidad de fertilizantes son críticas para aumentar la producción. Es también crucial que la intensificación de la producción se haga de la manera más sostenible posible, por ejemplo, limitando las dosis de fertilizantes a las necesidades de los cultivos.

La gestión de la fertilización es crítica para una producción agrícola eficiente y la limitación de efectos adversos sobre el medio ambiente. Sin embargo, no hay muchas herramientas sencillas para esta tarea en el mercado.

Fertilicalc Mobile

Fertilicalc es un programa de Windows que calcula las necesidades de nutrientes y fertilizantes para cultivos, así como la combinación más económica de fertilizantes comerciales.

Pretende ser una herramienta útil para agricultores en la estimación de necesidades de nutrientes y en la selección de fertilizantes. También tiene un propósito educativo, ayudando a estudiantes a entender mejor la lógica detrás de la gestión de la fertilización.

En muchos países emergentes o en desarrollo, el único acceso a Internet de muchas personas se produce a través de un dispositivo móvil. Por tanto, hemos considerado particularmente valioso adaptar Fertilicalc a una plataforma móvil; esto permitiría alcanzar un número de usuarios mucho mayor. Además, son precisamente los usuarios en países en desarrollo, donde una parte importante de la población vive de la agricultura, los que más podrían beneficiarse de una herramienta como esta.

Este proyecto pretende **adaptar *Fertilicalc* a una aplicación móvil** (que llamamos *Fertilicalc Mobile*) sencilla y funcional. Para integrar la funcionalidad de la aplicación original de escritorio, usaremos un script de Python[1] que realiza los cálculos necesarios a partir de la entrada del usuario. Instalaremos este script en un servidor HTTP, que expondrá el script mediante una API REST para que la aplicación pueda usarlo.

Asimismo, la aplicación consulta a la API Flora Codex, que proporciona información botánica sobre cultivos, la cual incluimos para reforzar el componente educativo.

Para construir la aplicación, hemos escogido el *framework* multiplataforma Flutter. Flutter proporciona rendimiento nativo tanto en Android como iOS, lo que nos permite maximizar al mismo tiempo la base de usuarios a la vez que la eficiencia del desarrollo.

Desarrollo del proyecto

Este proyecto tuvo una duración estimada de 1 año. La siguiente figura muestra el plan de trabajo seguido:

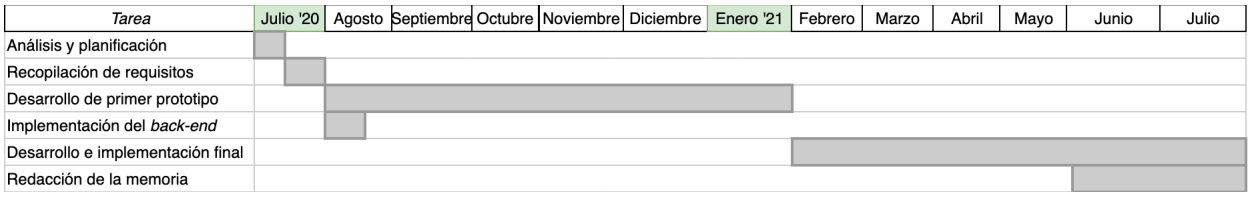


Figura 1-1. Plan de trabajo

[1] Este script fue proporcionado por el autor de Fertilicalc para escritorio. Nuestra labor en este aspecto ha sido integrarlo en la API REST e instalar el servidor HTTP (descrito en el apéndice A de la memoria completa).

Recursos

Los siguientes fueron los dispositivos móviles usados para probar la aplicación a lo largo de su desarrollo: *OnePlus 6* (Android 11), *Samsung Galaxy S9* (Android 10) y *Samsung Galaxy S6 Edge* (Android 7).

Para las pruebas en iOS, se usó el simulador de *XCode* para ejecutar la app en un *iPhone 12 Pro Max* virtual. A continuación, incluimos capturas de la ejecución en distintos dispositivos; de izquierda a derecha: *Samsung Galaxy S6 Edge*, *OnePlus 6*, y *iPhone 12 Pro Max* (virtual).

Figura 1-2. Ejecución de la aplicación en distintos dispositivos

Estructura de la memoria

A continuación, incluimos una descripción general de los distintos capítulos de la memoria:

1. **Introducción (*Introduction*).** Primera aproximación del problema, su contexto, y la naturaleza de la solución desarrollada.
2. **Marco de desarrollo (*Development framework*).** Ventajas del desarrollo multiplataforma, comparativa de *frameworks*, y conceptos fundamentales de Flutter, útiles para comprender la arquitectura y diseño de la aplicación.
3. **Requisitos (*Requirements*).** Descripción de los objetivos y funcionalidad de la aplicación.
4. **Arquitectura (*Architecture*).** Introducción a la gestión de estado con *Provider*, y descripción del patrón arquitectónico implementado por nuestra aplicación.
5. **Diseño e Implementación (*Design and Implementation*).** Descripción de la solución desarrollada, detallando los diferentes módulos y la implementación de los requisitos.
6. **Conclusiones y Líneas de mejora (*Conclusions and Future Work*).** Reflexión sobre el cumplimiento de los objetivos, y propuesta de mejoras futuras.

En el presente documento resumimos aspectos esenciales de la memoria completa, sin incluir ni detallar todo lo expuesto en esta.

2 MARCO DE DESARROLLO

En este capítulo, motivamos nuestra decisión de escoger el *framework* multiplataforma Flutter como nuestro marco de desarrollo. Después, introducimos conceptos básicos de Flutter que sirven como base para comprender la arquitectura, diseño, e implementación de nuestro proyecto.

2.1 Desarrollo multiplataforma

En la mayoría de los casos, los desarrolladores pretenden llegar a la audiencia más amplia posible con su aplicación; para garantizarlo, la aplicación idealmente debe lanzarse en múltiples plataformas. Si el equipo de desarrollo elige el desarrollo nativo, deberá mantener múltiples bases de código; esto conduce a más mantenimiento, más esfuerzo y, en última instancia, más coste.

El desarrollo multiplataforma es más rentable, ya que nos permite desplegar nuestro código en diferentes plataformas utilizando una sola herramienta SDK (Kit de Desarrollo de Software), mientras se logra un rendimiento similar al de una aplicación nativa. Solo se crea una base de código, por lo que el tiempo de desarrollo es menor y se mejora la reutilización del código. Además, la experiencia de usuario es más uniforme y está mejor alineada entre plataformas.

Nuestra aplicación está especialmente orientada a usuarios de países en desarrollo, donde la cuota de mercado de Android es muy superior a la de iOS. Sin embargo, consideramos que la producción de una versión para iOS refuerza nuestro objetivo de llegar a la mayor base de usuarios posible.

Según una encuesta global de desarrolladores de 2021, el 42% de los desarrolladores de multiplataforma utiliza Flutter, mientras que el 38% utiliza React Native.

Flutter ha mostrado un crecimiento sustancial desde su lanzamiento, con su adopción creciendo más rápido que la de React Native a lo largo de los últimos 4 años. Ha sido respaldado por el equipo de Google desde sus inicios y cuenta con una excelente documentación y una comunidad de desarrolladores muy dinámica.

Generalmente se cree que las aplicaciones multiplataforma presentan un rendimiento más pobre que las nativas. Sin embargo, la diferencia puede considerarse insignificante, especialmente para aplicaciones pequeñas y medianas. Flutter, en concreto, muestra un mejor rendimiento que React Native.

En resumen, hemos elegido Flutter debido a su creciente popularidad, su rendimiento fluido y su amplia documentación y soporte. A lo largo del desarrollo del proyecto, hemos demostrado beneficios adicionales, como la simplicidad y modularidad del ecosistema de *widgets* (explicado más adelante) y la amplia gama de paquetes externos creados por la comunidad de Flutter. Esto último nos ha permitido crear algunas partes de la aplicación especialmente rápido e implementar funciones que de otro modo habrían sido demasiado laboriosas.

2.2 Fundamentos de Flutter

Flutter es un SDK de código abierto creado por Google. Se utiliza para crear aplicaciones compiladas de forma nativa para dispositivos móviles (iOS y Android), web, y escritorio a partir de una única base de código. Flutter proporciona una forma sencilla de componer interfaces de usuario, centrándose en la apariencia, el comportamiento y la integración entre ambos.

Las aplicaciones de Flutter están escritas en el lenguaje *Dart*. También desarrollado por Google, es un lenguaje orientado a objetos que se enfoca en el desarrollo *frontend*.

Una de las principales diferencias de Flutter respecto a otros entornos como Android es que su UI (interfaz de usuario) está integrada en código, no en un archivo XML o similar. Flutter es un *framework* "declarativo".

En un *framework* "imperativo" (por ejemplo, Android, iOS o web), la UI se llama explícitamente en el código. En un *framework* "declarativo" como Flutter, el código declara que la UI debe verse de cierta manera, dado un cierto estado.

La idea central de la arquitectura es que la UI está construida a base de widgets. Casi todo en Flutter es un widget; se utilizan tanto para elementos de la interfaz (p. ej., `Text` –texto– o `Image` –imagen–) como para la disposición de los elementos (p. ej., `Row` –fila– o `Column` –columna–).

Un widget describe cómo debería verse según su configuración y estado actuales. Cuando su estado cambia, el widget reconstruye su descripción y el *framework* lo compara con la descripción anterior para realizar los cambios mínimos necesarios para la transición de un estado al siguiente. El trabajo principal de un widget es implementar un método `build()` que describe el widget en términos de otros widgets de nivel inferior (sus descendientes), que son construidos por el *framework* para representar la UI en pantalla.

Al escribir una aplicación, es común crear nuevos widgets que son subclases de `StatelessWidget` o `StatefulWidget`, dependiendo de si el widget administra algún estado.

Un **Stateless Widget** es un componente simple de UI que muestra solo los datos que se le proporcionan; no tiene "memoria". Un **Stateful Widget**, en cambio, es aquel con el que el usuario puede interactuar y que tiene "memoria". Su estado se almacena en un objeto *State*; cuando cambia, el objeto *State* llama a `setState()` y le dice al *framework* que reconstruya el widget en pantalla.

El *framework* proporciona una gama de widgets básicos; algunos de los más utilizados son `Text`, `Row`, `Column`, `Container`, etc. Además, se pueden incorporar widgets de paquetes externos.

Una aplicación de Flutter es esencialmente un árbol de widgets anidados. Cada pantalla se crea componiendo widgets para construir widgets más complejos. Los widgets que controlan disposición, como `Row` o `Column`, se encargan de organizar, restringir y alinear los widgets visibles que contienen.

Los widgets de Flutter incorporan diferencias críticas de plataforma como el desplazamiento, la navegación, los íconos y las fuentes para proporcionar un rendimiento nativo completo tanto en iOS como en Android.

A continuación, mostramos un ejemplo de una aplicación de Flutter:

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Welcome to Flutter'),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

Ejemplo de aplicación de Flutter

La clase que contiene la aplicación (`MyApp`) extiende `StatelessWidget`. El widget `Scaffold` proporciona una barra superior, y su propiedad `body` contiene el árbol de widgets para la pantalla de inicio. En este caso, encapsula un widget `Center`, que alinea a su hijo con el centro de la pantalla. Su hijo, un widget `Text`, muestra el texto correspondiente en la pantalla.

A continuación, se muestra el árbol de widgets para esta pantalla y su apariencia en dispositivos Android y iOS:

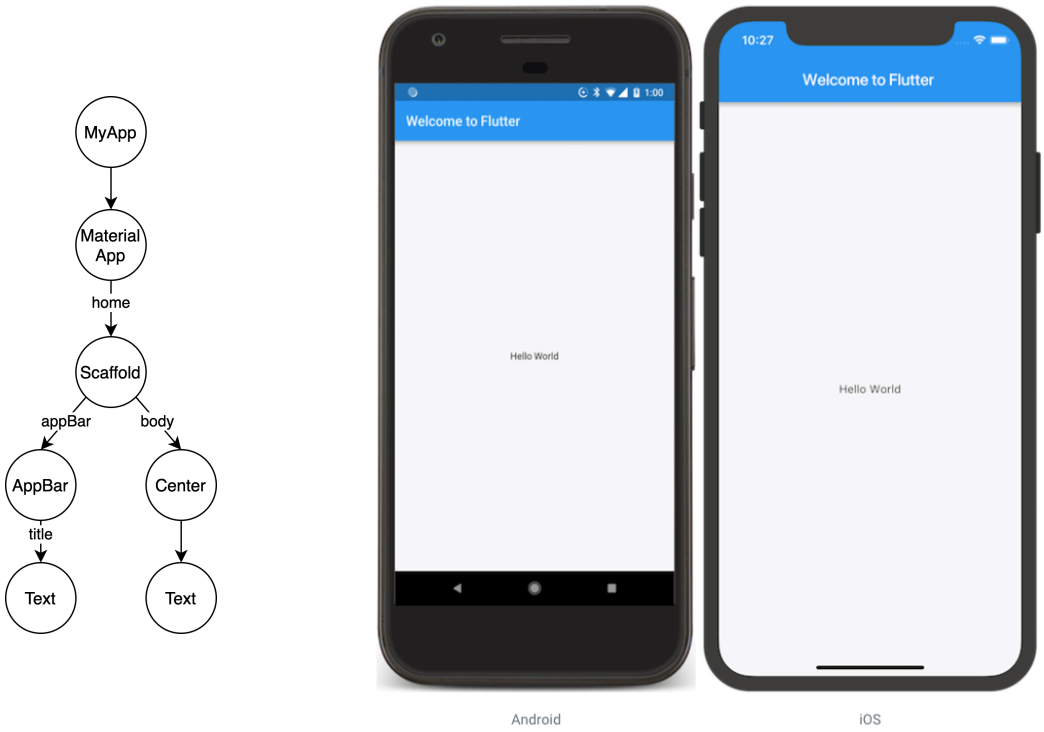


Figura 2-1. Árbol de widgets y apariencia de la aplicación de ejemplo

El árbol de widgets puede volverse bastante complejo rápidamente; por esta razón, en nuestro proyecto, dividimos las pantallas más complejas en múltiples widgets. Por ejemplo, para crear la pantalla que muestra la lista de cultivos, definimos esta en términos de widgets más pequeños (ver sección 5.2.2).

También aprovecharemos la modularidad del ecosistema de widgets para mejorar la reutilización y limpieza del código. Dos ejemplos son `CustomAppBar` y `CustomFloatingButton`, widgets que hemos modificado y que reutilizamos en diferentes pantallas:

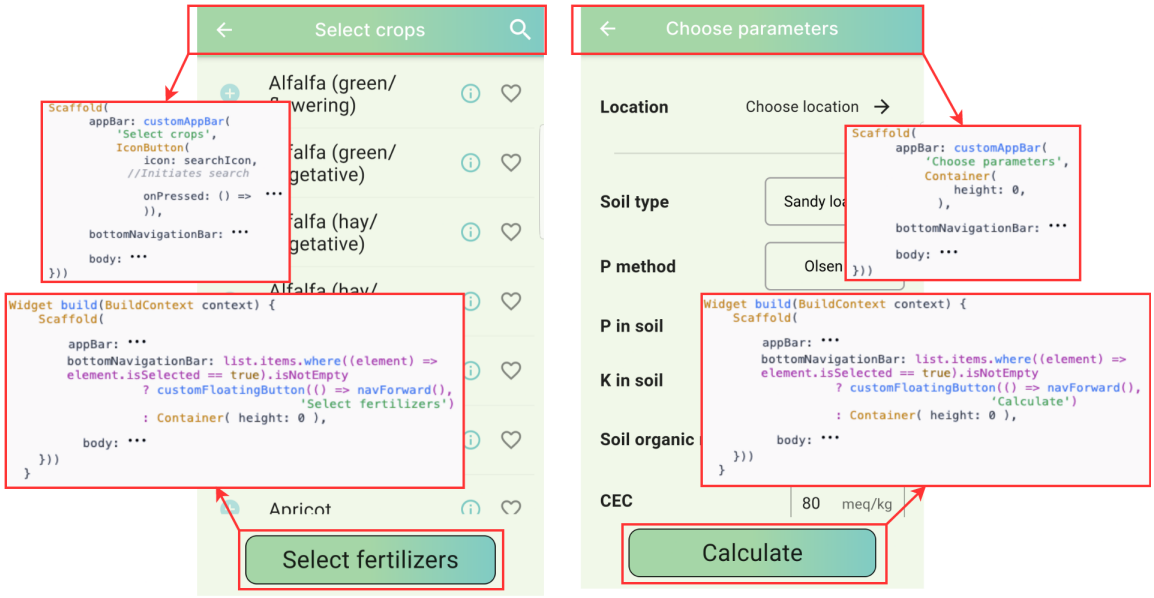


Figura 2-2. Widgets *CustomAppBar* y *CustomFloatingButton*

3 REQUISITOS

En nuestro proyecto, la recopilación de los requisitos fue un proceso iterativo que evolucionó en paralelo con el diseño y la implementación. Algunos de los requisitos funcionales se agregaron una vez el desarrollo estaba en marcha; en nuestra memoria describimos los que se incluyeron en el producto final.

Nótese que la esencia de la aplicación y, en consecuencia, la mayoría de los requisitos, son el resultado de adaptar la funcionalidad de la aplicación de escritorio *Fertilicalc* al entorno móvil.

Descripción básica de la aplicación

La aplicación permite al usuario seleccionar algunos cultivos, junto con algunos productos fertilizantes que aplicar a estos, y personalizar algunos parámetros, como su ubicación o propiedades del suelo. A partir de estos datos de entrada, el usuario obtendrá las dosis de fertilizante que aplicar a cada cultivo, y otros datos útiles para la gestión de la fertilización.

Características del usuario

La aplicación está destinada principalmente a agricultores, especialmente en países en desarrollo. También está dirigida a profesionales y estudiantes de agricultura de todo el mundo.

Un usuario típico debe contar con experiencia básica en el uso de aplicaciones y sistemas operativos móviles, y tener un conocimiento intermedio de terminología agrícola, para utilizar la aplicación eficazmente.

Objetivos del usuario

La aplicación permite al usuario realizar un cálculo para determinados cultivos con determinadas características.

Para este cálculo, el usuario agrega cultivos, junto con su rendimiento, luego agrega fertilizantes para cada uno de estos cultivos. El usuario puede también personalizar el cálculo con datos como la ubicación, el tipo de suelo o el suministro de agua. Cuando se ha realizado el cálculo, la aplicación muestra la dosis de cada fertilizante elegido que se debe aplicar a cada cultivo, junto con otra información pertinente, como las necesidades de nutrientes.

El usuario puede guardar estos resultados y verlos más tarde, así como compartirlos fuera de la aplicación.

Requisitos funcionales

R-01 – En una de las pantallas, se muestra una lista de cultivos, cada uno identificado por su nombre y acompañado por al menos un botón, que se pulsa para seleccionar el cultivo.

R-02 – Al seleccionar un cultivo, el usuario puede agregarlo al cálculo introduciendo un rendimiento (kilos por hectárea *–kg/Ha–* de la cosecha) positivo.

R-03 – Al seleccionar un cultivo, el usuario puede personalizar dos parámetros: 'porcentaje de residuos' (entero) y 'quema de residuos' (booleano).

R-04 – El usuario puede buscar un cultivo específico mediante una barra de búsqueda.

R-05 – El usuario puede marcar cualquier cultivo como favorito y luego desmarcarlo. La selección de favoritos persiste entre ejecuciones.

R-06 – Para cada cultivo de la lista, la aplicación puede mostrar la siguiente información: especie, género, familia, y una foto del cultivo.

R-07 – En una de las pantallas, se muestra una lista de productos fertilizantes disponibles, cada uno identificado por un nombre y acompañado por al menos un botón, que se pulsa para seleccionarlo.

R-08 – El usuario puede comenzar a seleccionar fertilizantes cuando haya agregado al menos un cultivo.

R-09 – Al seleccionar un fertilizante, el usuario puede editar los parámetros booleanos 'incorporado' y 'basal'.

R-10 – Al seleccionar un fertilizante, el usuario debe seleccionar a qué cultivos aplicarlo.

R-11 – El usuario puede buscar un fertilizante específico mediante una barra de búsqueda.

R-12 – El usuario puede marcar cualquier fertilizante como favorito y luego desmarcarlo. La selección de favoritos persiste entre ejecuciones.

R-13 – El usuario puede elegir la ubicación para el cálculo.

R-14 – Una vez el usuario elige una ubicación, esta se guarda como predeterminada y persiste entre ejecuciones.

R-15 – El usuario puede personalizar los siguientes parámetros globales para el cálculo: 'tipo de suelo' (4 valores posibles), 'P en el suelo' (entero), 'suministro de agua' (2 valores posibles), 'labranza' (booleano), etc.

R-16 – La aplicación muestra inicialmente valores predeterminados para los parámetros globales. Cuando el usuario cambia estos, los nuevos valores se guardan y persisten entre ejecuciones.

R-17 – Se pueden restaurar los valores predeterminados de los parámetros globales.

R-18 – El usuario puede activar el cálculo de resultados solo cuando se hayan agregado al menos un cultivo y un fertilizante, y se haya elegido una ubicación.

R-19 – Una vez que se han calculado los resultados, la aplicación los muestra en otra pantalla.

R-20 – Los resultados mostrados para un cálculo incluyen, para cada cultivo de la rotación: la dosis de cada fertilizante elegido para el cultivo, acidificación, pérdidas de nitrógeno, etc.

R-21 – El usuario puede guardar los resultados de un cálculo para verlos posteriormente.

R-22 – En una de sus pantallas, la aplicación presenta un historial de resultados guardados previamente.

R-23 – El usuario puede consultar cualquier resultado del historial, accediendo a toda la información correspondiente a ese resultado.

R-24 – El usuario puede compartir resultados (ya sean recién calculados o del historial) fuera de la aplicación.

R-25 – El usuario puede eliminar resultados específicos del historial.

R-26 – El usuario puede borrar el historial completo de resultados.

R-27 – El usuario puede eliminar todos los datos almacenados, p.ej., su ubicación, favoritos, etc.

R-28 – En una de sus pantallas, la aplicación permite al usuario cambiar el idioma de la aplicación.

Requisitos no funcionales

- **Rendimiento.** La aplicación debe tener un tamaño inferior a 20 MB.
- **Usabilidad.** La dificultad del usuario para comprender la aplicación debe ser de fácil a intermedia. Para ayudar en este aspecto, se debe incluir un tutorial en formato visual.
- **Internacionalización.** La aplicación estará disponible en 7 idiomas: inglés, español, italiano, ruso, hindi, árabe y chino. Cuando se configura en un idioma determinado, todos los textos de la aplicación se mostrarán en el idioma elegido, excepto algunos términos técnicos específicos. Cuando se ejecuta en árabe, la interfaz de usuario tendrá una presentación de derecha a izquierda.
- **Compatibilidad.** La aplicación se podrá ejecutar en versiones de Android más recientes que 5.0 y versiones de iOS más recientes que 12.4. La interfaz de usuario debe mostrarse adecuadamente en dispositivos con un tamaño de pantalla entre 5,1 pulgadas (*Samsung Galaxy S6 Edge*) y 6,68 pulgadas (*iPhone 12 Pro Max*).

4 ARQUITECTURA

La arquitectura define el esqueleto de una aplicación, en el que se basan los diferentes elementos estructurales y su relación con el marco de desarrollo. En este capítulo, describimos nuestra elección de herramienta para manejar la *gestión de estado*, un aspecto central de la arquitectura. Luego, proponemos un patrón arquitectónico alrededor del cual se construirá la aplicación.

4.1 Gestión de estado

En Flutter, la *gestión de estado* se refiere a la forma en que una aplicación maneja la distribución del estado a través del árbol de widgets. Define cómo las diferentes partes de la aplicación interactúan entre sí y cómo se almacenan y se accede a los datos mutables.

Flutter no impone ningún tipo de arquitectura o solución para la gestión de estado; esto ha llevado a la creación de múltiples soluciones, y un puñado de enfoques arquitectónicos han surgido de la comunidad.

Decidimos elegir *Provider* para este proyecto debido a su simplicidad, baja curva de aprendizaje, amplia documentación y soporte del equipo de Flutter. Dado que nuestra aplicación no es particularmente grande (no incluye muchas pantallas o un gran conjunto de funciones), consideramos que esto era suficiente y preferible a un enfoque más sofisticado (p.ej., *Redux* o *BLoC*).

4.2 Provider

En Flutter, para que el estado sea accesible a múltiples widgets, "elevamos el estado" a un ancestro de todos los widgets que usan ese estado común.

Flutter cuenta con mecanismos para que los widgets proporcionen datos y servicios a sus descendientes. *Provider* utiliza estos widgets de bajo nivel, pero es más sencillo de implementar. Los siguientes son tres componentes esenciales de *Provider*:

- ***ChangeNotifier*:** Clase que proporciona notificación de cambios a sus oyentes. Es una forma de encapsular el estado de la aplicación, a través de "**modelos**" (objetos que encapsulan datos). Puede haber varios *ChangeNotifier*, cada uno para un modelo específico. A continuación, se muestra un extracto de `CropListModel`, uno de los modelos que definimos en nuestra aplicación:

```
class CropListModel extends ChangeNotifier {
  List<Crop> items;
  CropListModel({this.items});

  //Selects a crop from the list or unselects it
  void toggleSelected(int index) {
    items[index].isSelected = !items[index].isSelected;
    notifyListeners();
  }

  //Marks a crop as fav or unmarks it
  void toggleFav(int index) {
    items[index].fav = !items[index].fav;
    notifyListeners();
  }
      ...
```

Clase *CropListModel*

Este modelo almacena la lista de cultivos (objetos `Crop`) que mostrará la aplicación y define los métodos que se llaman cuando el usuario selecciona un cultivo específico o lo marca como favorito. El único código específico de *ChangeNotifier* es el método `notifyListeners()`, que se llama cada vez que

el modelo cambia de una manera que podría cambiar la UI. En este caso, la acción de seleccionar o deseleccionar un cultivo de la lista (llamando al método `toggleSelected`) se reflejará en la UI.

La siguiente figura muestra un fragmento de la pantalla que muestra la lista de cultivos. Inicialmente, ningún cultivo está seleccionado; cuando el usuario pulsa el botón, se llama al método `toggleSelected`, que a su vez llama a `notifyListeners`. Esto activa una reconstrucción de la UI de cualquier widget que estuviera escuchando; en este caso, se reflejará cambiando el ícono a la izquierda. La misma secuencia se repite al deseleccionar el cultivo.

Figura 4-1. Actualización de la UI con *notifyListeners*

- ***ChangeNotifierProvider***: Widget que proporciona una instancia de un *ChangeNotifier* a sus descendientes. Para proporcionar más de una clase, utilizamos el widget `MultiProvider`. Es común colocar este widget en la raíz del árbol de widgets de la aplicación, para que todos los widgets que lo necesiten puedan acceder a él.

  A continuación, mostramos un extracto del archivo *main.dart* de nuestra aplicación, donde `MultiProvider` expone los datos de nuestros diferentes modelos:

```
void main() => runApp(MyApp());

...
class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext _) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (c) => CartModel()),
        ChangeNotifierProvider(create: (c) => CropListModel()),
        ChangeNotifierProvider(create: (c) => FertListModel()),

 ...
```

Uso de *MultiProvider* en *main.dart*

- ***Consumer***: Widget que accede a los datos del *ChangeNotifier*. Debemos especificar el tipo de modelo al que queremos acceder; en nuestro ejemplo, queremos `CropListModel`, por lo que escribimos `Consumer <CropListModel>`. Este código accede al modelo para construir la lista de cultivos en pantalla:

```
return Consumer<CropListModel>(
      builder: (context, list, child) => ListTile(
          leading: (list.items[index].isSelected
              //Shows 'Check' button if it is selected
              ? IconButton(
                  onPressed => list.toggleSelected(index);
                  icon: checkIcon,
                )
              //Shows 'Add' button that triggers dialog if not selected
              : IconButton(
                  onPressed => list.toggleSelected(index);
                  icon: addIcon,
                )),
```

Uso de *Consumer<CropListModel>*

`builder` es una función que se llama cada vez que cambia el *ChangeNotifier*; cuando se llama a `notifyListeners()`, se llama a los métodos `builder` de todos los *Consumer* correspondientes. En el código anterior, un operador ternario verifica el valor del campo `isSelected` para determinar qué icono mostrar para un cultivo específico (como se ilustra en la figura 4-1).

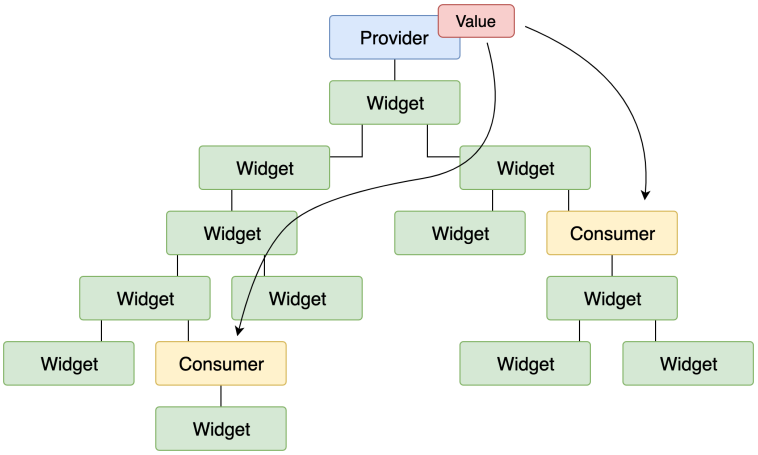


Figura 4-2. Gestión de estado con *Provider*

Esencialmente, *Provider* es un widget que hace que algún valor, como el estado de un *modelo*, esté disponible para los widgets inferiores. Un widget *Consumer* escucha los cambios en el valor y reconstruye los widgets inferiores cuando ocurren cambios.

4.3 Arquitectura de la aplicación

A medida que un proyecto se vuelve más complejo, la necesidad de definir una arquitectura se vuelve esencial para mantener el código escalable y mantenible. Al final del desarrollo del primer prototipo de nuestra aplicación, esta necesidad se volvió imperativa y el código fue refactorizado para adaptarlo a una arquitectura estructurada.

Para nuestra aplicación, hemos seguido una arquitectura denominada '*Modelo –Vista - Comandos + Servicios*'. Esta divide la mayoría de las clases de nuestro código en 4 niveles o capas:

- **Comandos.** Tareas a nivel de aplicación (p. ej., cargar datos desde la memoria) que se encapsulan en una instancia de objeto y se inician con una llamada (p. ej., `LoadData().run()`). Los comandos se pueden iniciar desde la Vista, a raíz de eventos en la UI, o mediante otros comandos. Los comandos encapsulan su propio estado y pueden ser asíncronos.
- **Servicios.** Tareas que interactúan con el "mundo exterior", p.ej., Internet, el sistema de archivos local, etc. Además, analizan y devuelven cualquier dato que reciben. Nunca interactúan directamente con los Modelos; en su lugar, los Comandos realizarán llamadas a los Servicios para actualizar posteriormente los Modelos con los resultados.
- **Modelos.** Clases que mantienen el estado de la aplicación y encapsulan datos, notificando a los oyentes cuando los datos han cambiado. Proporcionan una API para acceder, filtrar y manipular estos datos.
- **Vista.** Widgets utilizados para construir las diferentes pantallas dentro de la aplicación. Junto con el código de la UI, contiene cierta lógica básica para "controlar" la vista. En algunos casos, la Vista se comunica directamente con el *Provider* para realizar tareas básicas. Por ejemplo, en el ejemplo de la figura 4-1, la Vista llama directamente al método `toggleSelected` del Modelo.

La ventaja crucial de esta arquitectura es que aprovecha al máximo *Provider* como enfoque de gestión de estado. Esto nos permite simplificar y modularizar el código, mejorando su manejabilidad y reutilización.

El siguiente diagrama ilustra cómo los diferentes módulos interactúan dentro de nuestra aplicación:

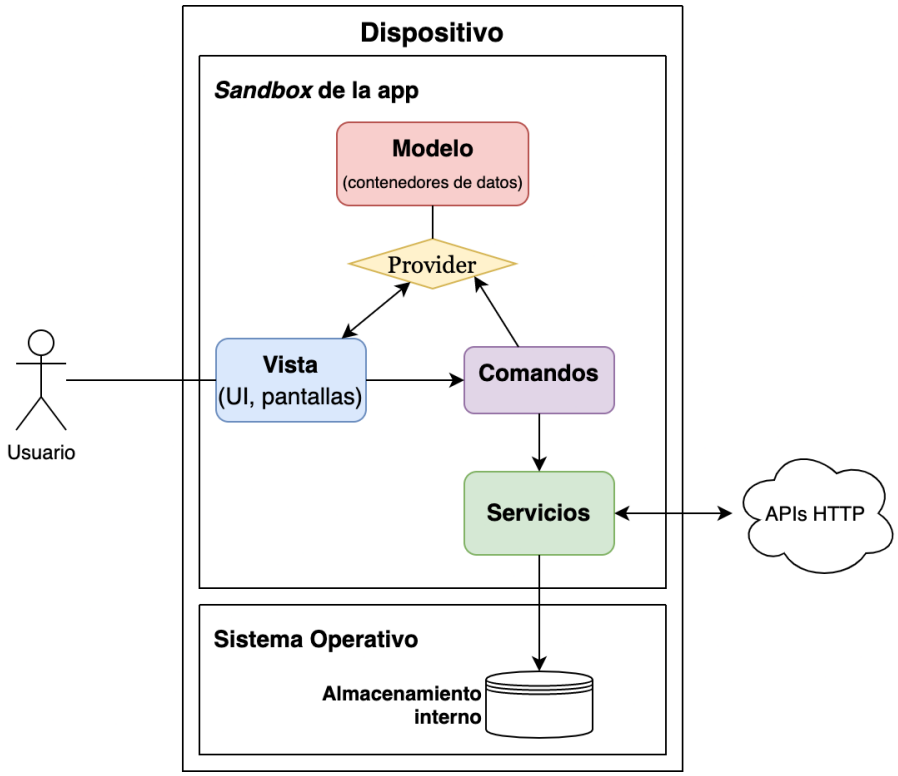


Figure 4-2. Integración de componentes arquitectónicos de la aplicación

Aquí vemos la característica central de la arquitectura: el uso de *Provider* como intermediario entre el Modelo y los otros módulos. Cuando la Vista recibe entradas del usuario, puede cambiar los datos del Modelo accediendo al *Provider*, ya sea directa o indirectamente a través de Comandos. Cuando se actualiza el Modelo, la Vista se actualiza directamente a través del *Provider*. Los Servicios, que interactúan con entidades externas, solo son accesibles directamente por los Comandos.

5 DISEÑO E IMPLEMENTACIÓN

El desarrollo de nuestra aplicación comenzó con la producción de un prototipo inicial que implementaba solo requisitos críticos. La facilidad que ofrece Flutter para construir la UI rápidamente desde cero hace que sea atractivo emplear un estilo de desarrollo iterativo; aprovechando esto, gradualmente se fueron agregando más requisitos, y la estructura se adaptó a la arquitectura presentada en el apartado 4.3. La evolución desde los primeros bocetos de pantallas hasta el diseño final se ilustra en el Apéndice B de la memoria completa.

En este capítulo, examinamos el diseño final de nuestra aplicación, cómo se implementan los diferentes módulos de nuestra arquitectura y cómo se integran para cumplir con los requisitos de nuestra aplicación.

5.1 Flujo de usuario

La base de nuestro diseño es el siguiente flujo de usuario:

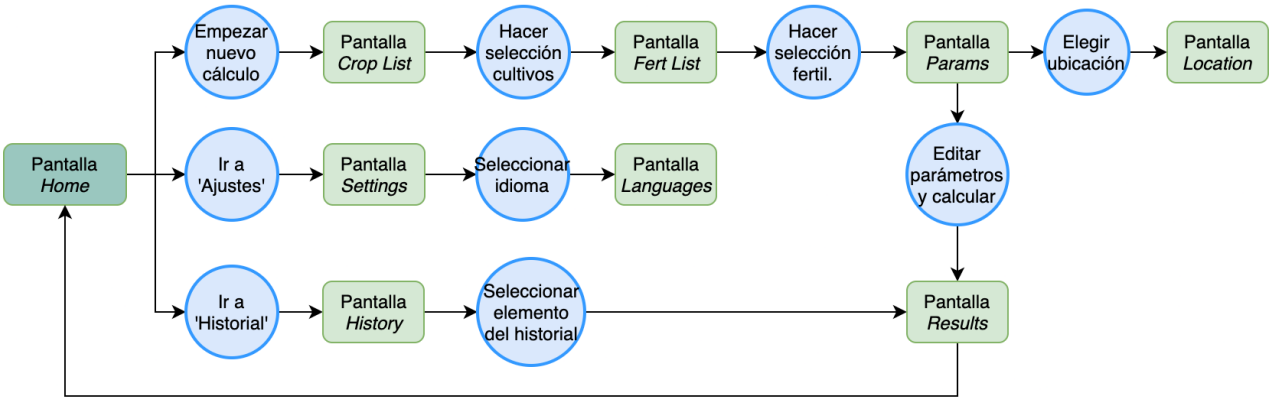


Figura 5-1. Descripción general del flujo de usuario

El punto de partida es la pantalla *Home*; a partir de aquí, se pueden tomar tres caminos diferentes:

- **Inicio de un nuevo cálculo.**
  - La aplicación muestra primero la pantalla *Crop List*.
  - Una vez el usuario realiza una selección de cultivos, la aplicación navega a la pantalla *Fert List*.
  - Una vez realizada la selección de fertilizantes, la aplicación navega a la pantalla *Params*. Cuando el usuario elige seleccionar una ubicación, la aplicación navega a la pantalla *Location.*
  - Una vez personalizados los parámetros globales, el usuario puede solicitar el cálculo de resultados, después de lo cual la aplicación navega a la pantalla *Results*.
- **Sección de ajustes.**
  - La aplicación muestra la pantalla *Settings*.
  - Si el usuario elige cambiar el idioma, se le dirige a la pantalla *Languages*.
- **Sección de historial.**
  - La aplicación muestra la pantalla *History*.
  - Si el usuario selecciona cualquiera de los elementos del historial, se le dirige a la pantalla *Results*, que se reutiliza de la ruta de nuevo cálculo.

5.2 Vista

En esta sección, discutimos cómo se implementan las diferentes pantallas a través del módulo *Vista* y mostramos cómo se ven estas pantallas en el dispositivo. Para mayor brevedad, solo mencionamos ciertos elementos de la *Vista* para ilustrar cómo se utilizan las diferentes herramientas proporcionadas por el *framework* y cómo se aplica la arquitectura propuesta.

El módulo *Vista* abarca código que describe la presentación de la interfaz de usuario: cómo debe verse, qué datos debe mostrar, cómo interactúa con el usuario, etc. También incluye código que realiza tareas como llamar a los comandos para realizar tareas más complejas, navegar a otras pantallas, personalizar los resultados de búsqueda, etc.

Pantalla *Home*

Esta pantalla es el punto de partida del flujo de usuario.[2] La siguiente figura ilustra las rutas de navegación que se derivan de ella:

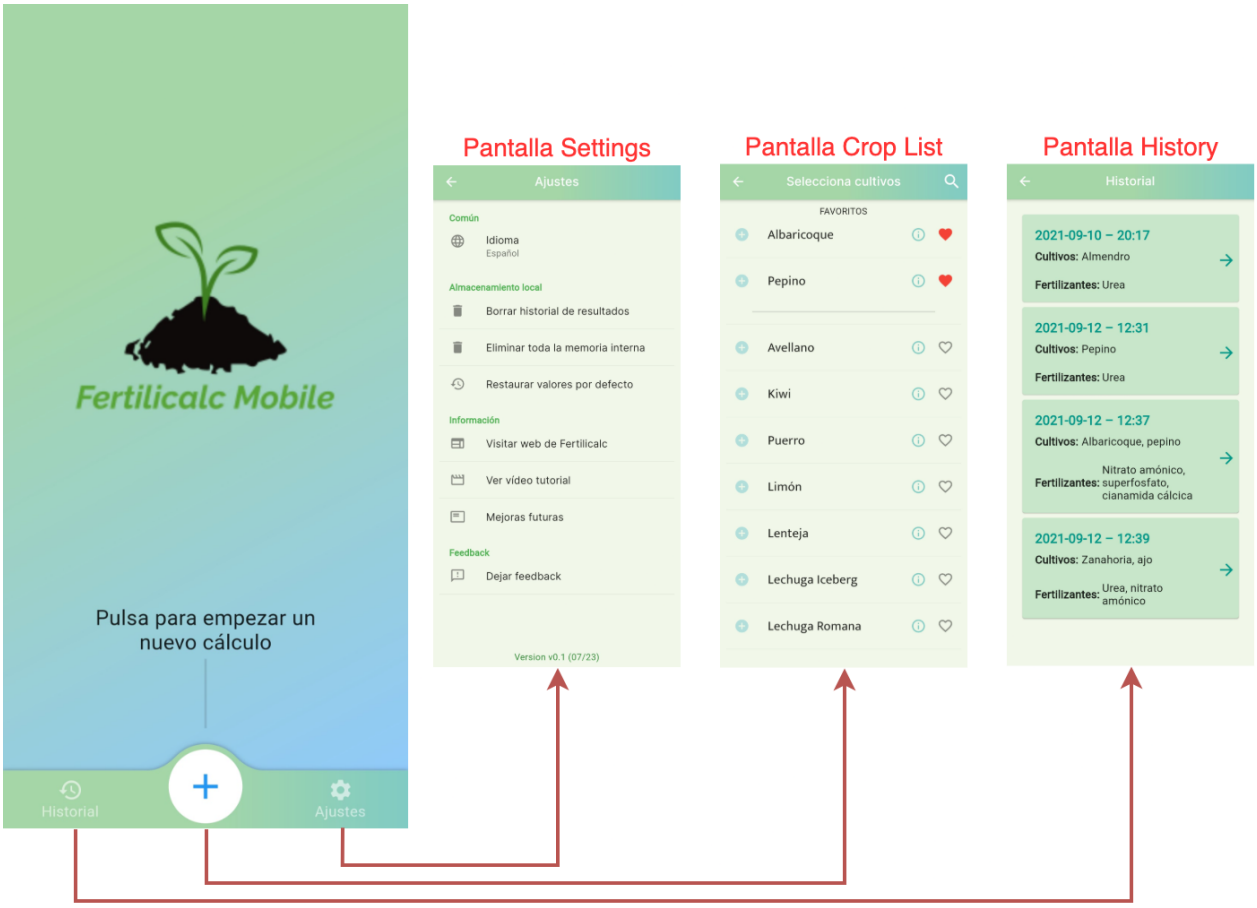


Figura 5-2. Navegación desde la pantalla *Home*

Pantalla *Crop List*

La pantalla *Crop List* está definida principalmente por 3 widgets: `CropListScreen`, `CropDialog` y `CropInfoSheet`. El primero construye la pantalla completa, mientras que los otros dos definen componentes adicionales que se integran con la pantalla.

[2] En la memoria completa se incluyen extractos comentados del código de las pantalla *Home* y *Crop List*

Las siguientes capturas muestran la apariencia de esta pantalla:

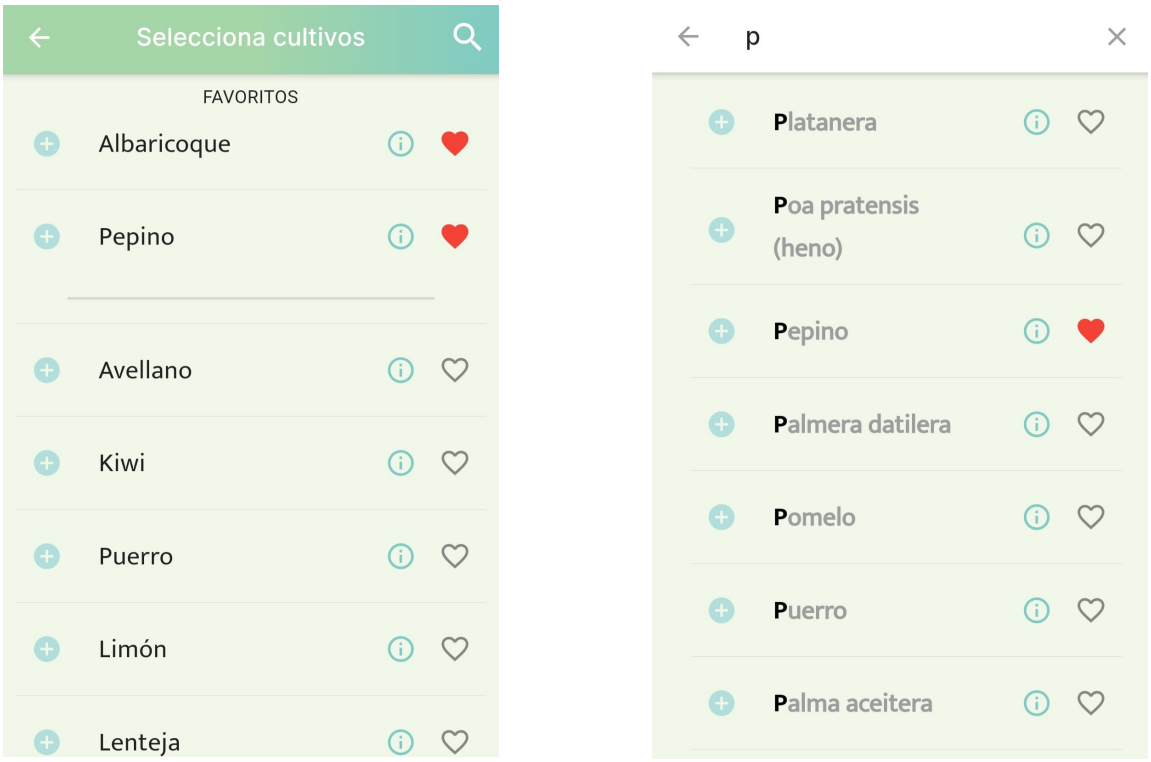


Figura 5-3. Pantalla *Crop List*

A continuación, se muestra el árbol de widgets para esta pantalla, donde podemos observar el anidamiento de nuestros widgets, y una captura de pantalla que ilustra la disposición de estos. El widget `cropListBody` contiene el `ListView` que crea la lista de parejas `CustomCropTile` - `Divider`.

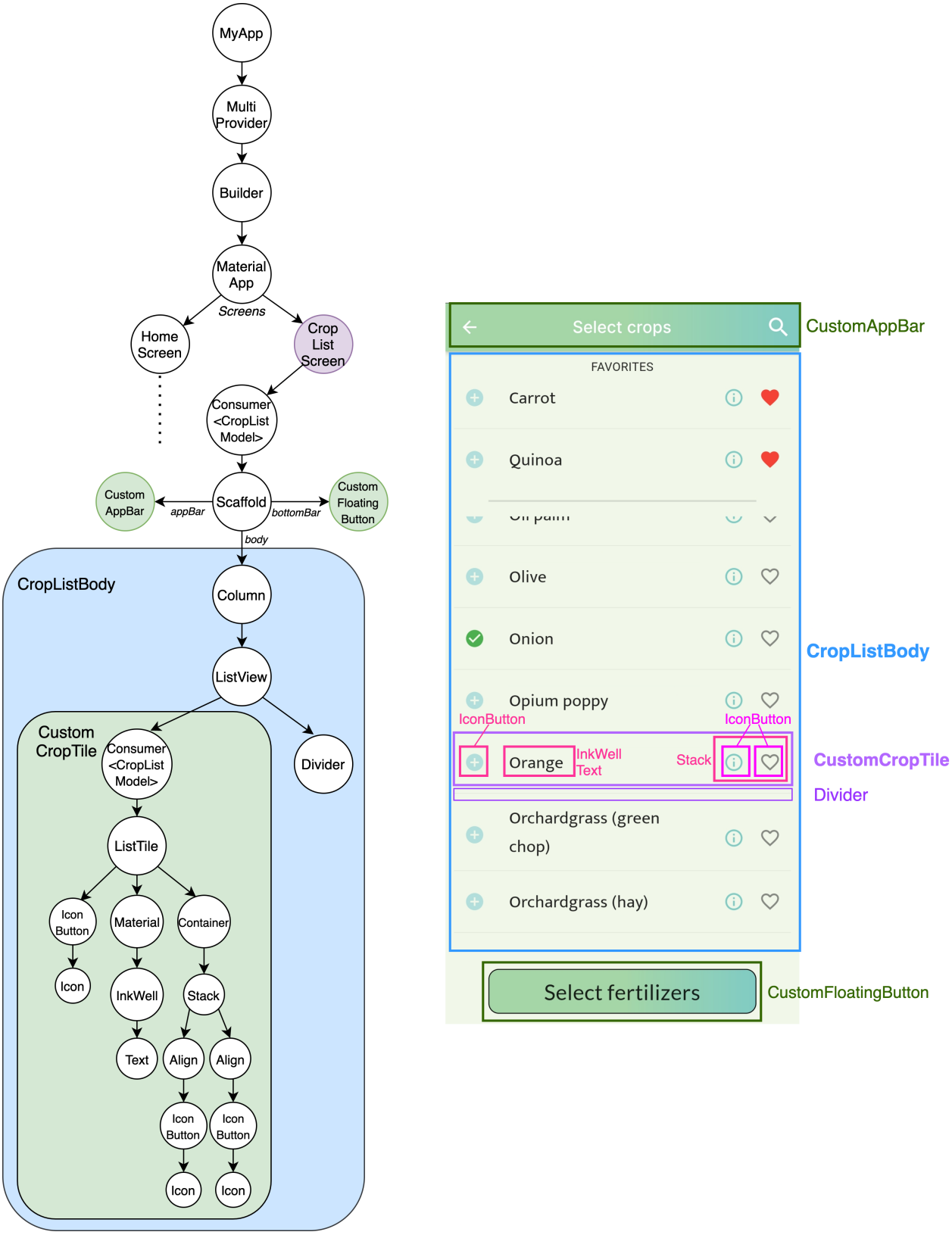


Figura 5-4. Árbol de widgets de la pantalla *Crop List*

El widget `CropDialog` invoca un cuadro de diálogo en el que el usuario puede ingresar un rendimiento para el cultivo y editar los dos parámetros definidos en R-03. Cuando el usuario ingresa un rendimiento positivo, el cultivo se puede agregar pulsando el botón verde.

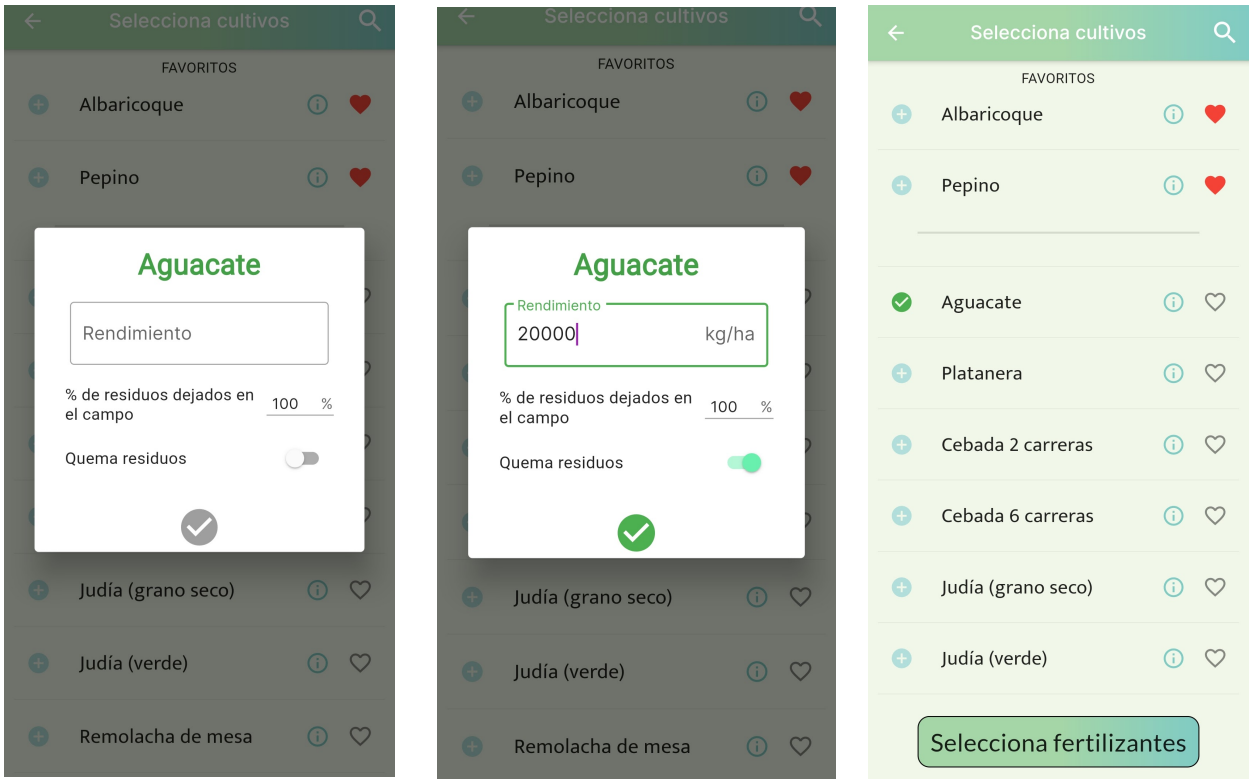


Figura 5-5. Adición de un cultivo mediante *Crop Dialog*

El widget `CropInfoSheet` muestra, para un cultivo dado: especie, género, familia y una imagen del cultivo (R-06):

Figura 5-6. *Crop Info Sheet*

Pantalla *Fert List*

Esta pantalla se crea de manera análoga a *Crop List*, ya que los comandos y las estructuras de datos con las que interactúa son muy similares. En las siguientes capturas podemos observar sus características:

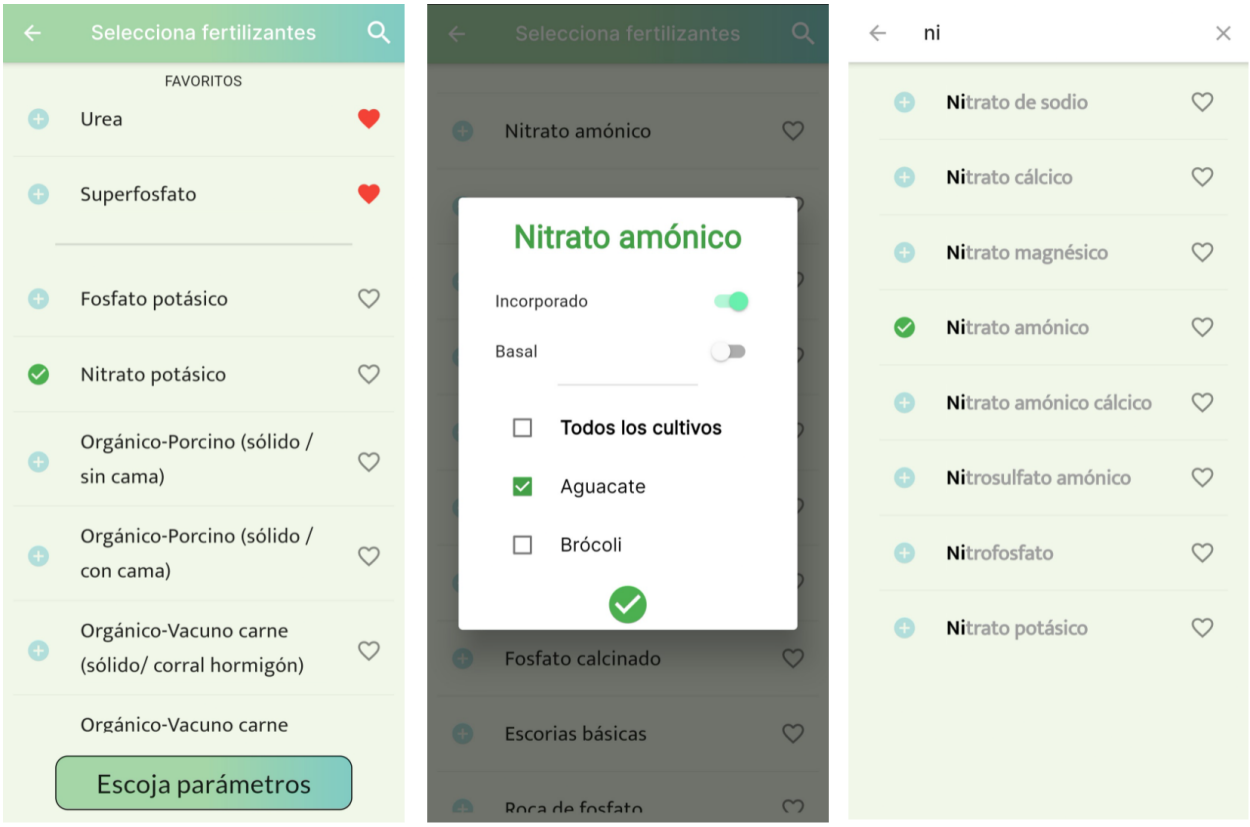


Figura 5-7. Pantalla *Fert List*

Pantalla *Params*

La pantalla *Params* permite al usuario elegir su ubicación y personalizar los parámetros globales del cálculo (R-13, R-15). En la parte superior de la pantalla se muestra o bien un mensaje pidiendo al usuario que elija una ubicación, o bien la última ubicación elegida por el usuario. El resto de la pantalla contiene una lista desplazable de parejas nombre–valor para cada parámetro global editable por el usuario.

Figura 5-8. Pantalla *Params*

Cuando el usuario solicita elegir una nueva ubicación, la pantalla invoca el paquete *Permission_handler*, que se encarga de solicitar los permisos de ubicación al usuario, adaptándose tanto a Android como a iOS.

Pantalla *Location*

Esta pantalla implementa el paquete *Google Maps Place Picker*, que presenta una interfaz del estilo de Google Maps donde el usuario puede ver su ubicación actual y elegir cualquier punto del mapa mundial. Hemos utilizado un paquete externo para externalizar esta funcionalidad ya que sería muy complejo hacerlo desde cero, especialmente porque implicaría interactuar con los *frameworks* subyacentes de iOS y Android.

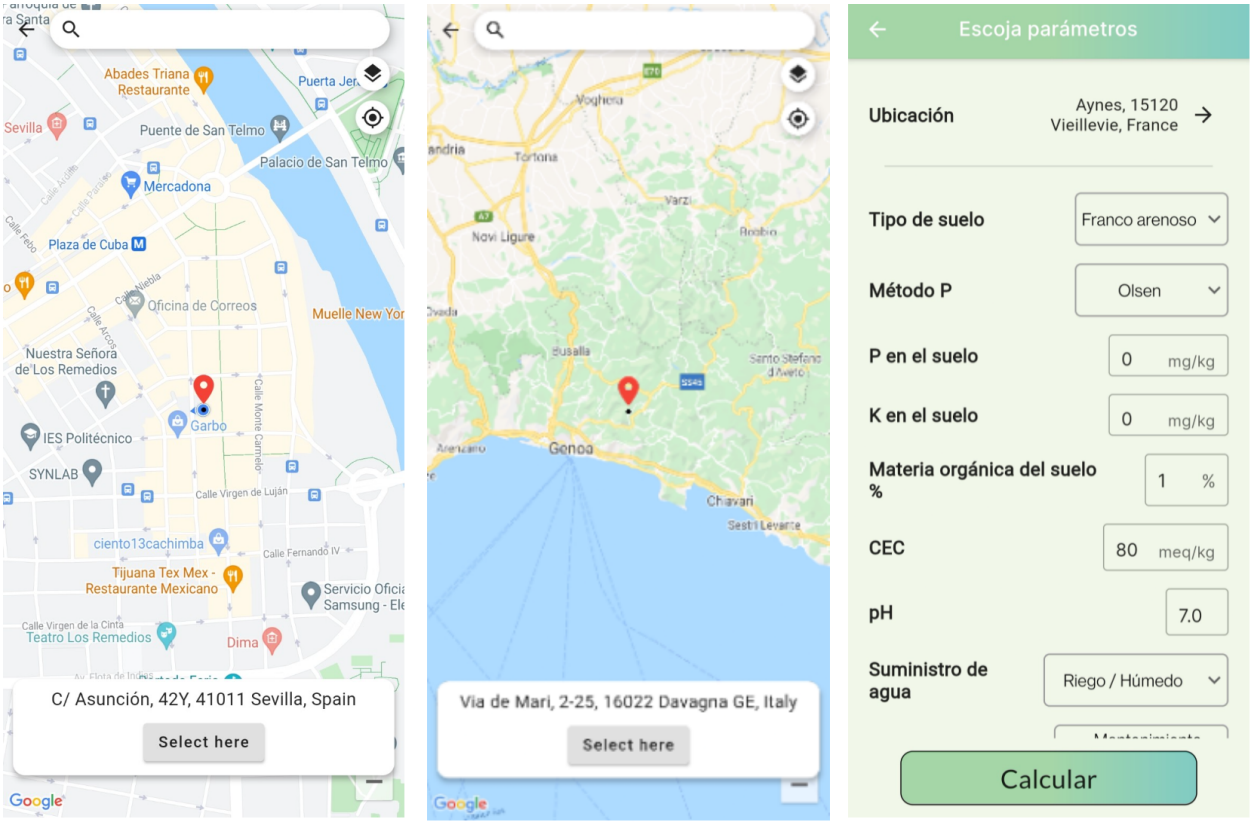


Figura 5-9. Pantalla *Location*

Pantalla *Results*

La pantalla *Results* muestra una ficha para cada cultivo involucrado en el cálculo. Cada ficha muestra los resultados del cultivo correspondiente. Inicialmente, la tarjeta solo muestra el rendimiento del cultivo y la dosis correspondiente a cada fertilizante. Cuando el usuario pulsa el botón de expansión, la tarjeta muestra también el resto de los parámetros definidos en R-20.

En la barra inferior de la pantalla, el usuario puede optar por guardar los resultados (R-21) o compartirlos con un tercero (R-24).

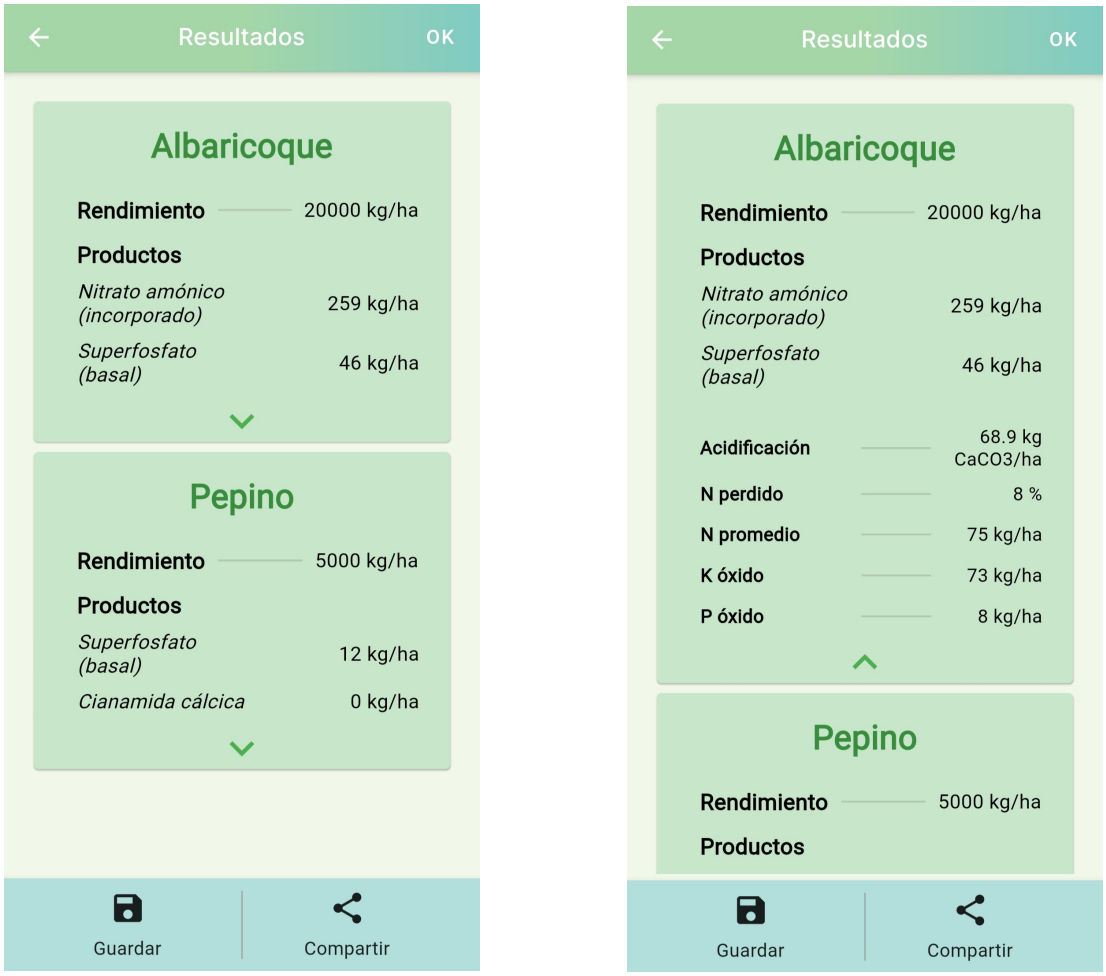


Figura 5-10. Pantalla *Results* (I)

Pantalla *History*

Esta pantalla muestra una lista de todos los cálculos que el usuario ha guardado (R-22). Cada cálculo se presenta con su fecha y hora, y los cultivos y fertilizantes involucrados. Al pulsar cualquiera de ellos, la aplicación navega a la pantalla *Results*, que muestra los resultados del cálculo elegido (R-23).

La pantalla *Results* en este caso solo difiere en la barra inferior, donde en lugar de la opción de guardar, el usuario tiene la opción de eliminar el resultado (R-25).

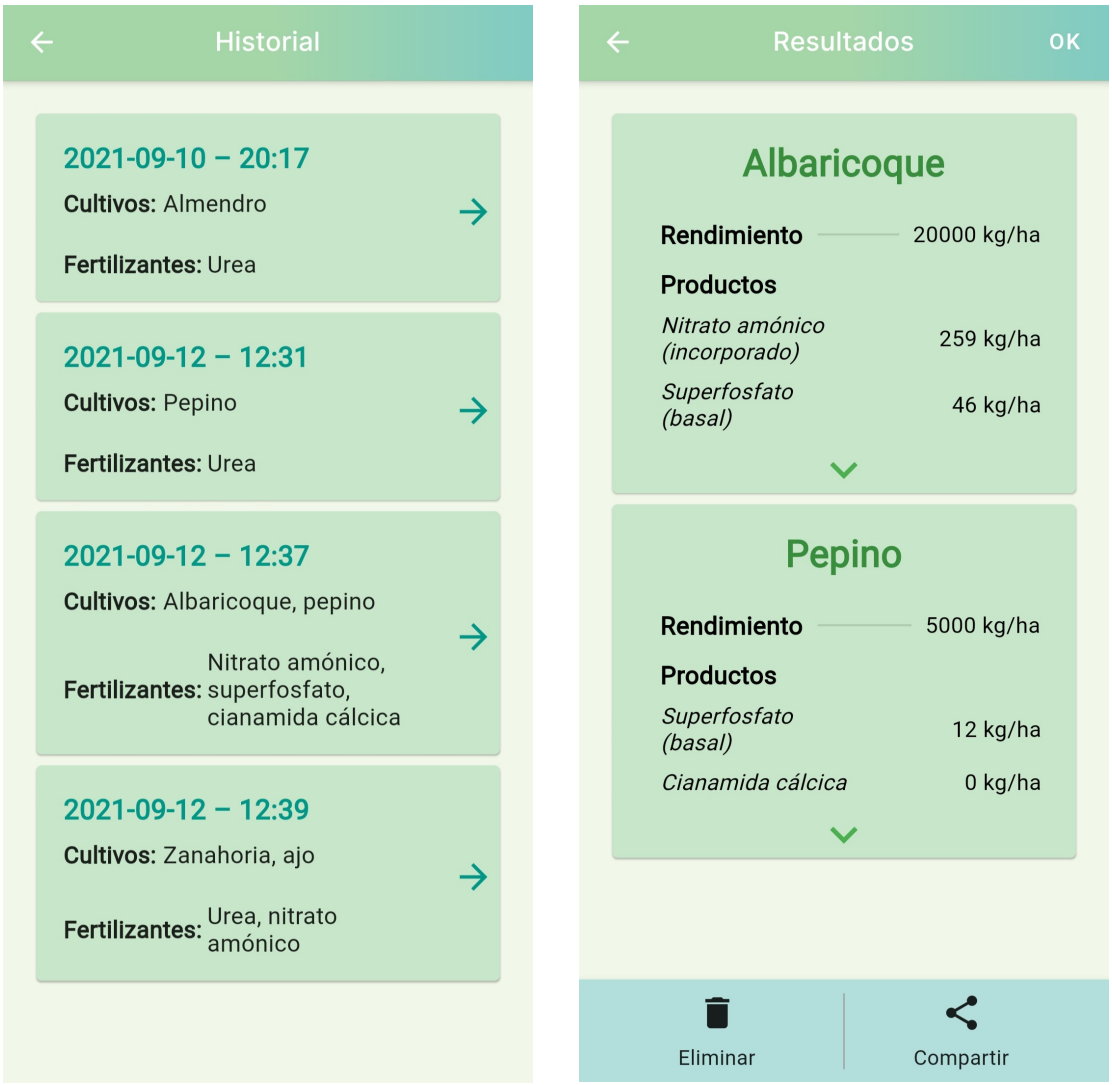


Figura 5-11. Pantalla *History* y pantalla *Results* (II)

Pantalla *Settings* y pantalla *Languages*

La pantalla *Settings* utiliza el paquete *Settings_UI* para mostrar una lista de secciones, que definimos como:

- ***Común.*** Incluye el elemento 'Idioma', que muestra el idioma actualmente seleccionado, y al pulsarse redirige al usuario a la pantalla *Languages*.
- ***Almacenamiento local.*** Incluye elementos que realizan tareas relacionadas con la persistencia de datos. El elemento "*Borrar historial de resultados*" llama al comando `deleteHistory` (R-26), el elemento "*Eliminar toda la memoria interna*" llama al comando `deleteAllData` (R-27), y el elemento "*Restaurar valores por defecto*" llama al comando `restoreDefault` (R-17).
- ***Documentación.*** Incluye elementos que brindan al usuario información adicional para comprender mejor la aplicación. El elemento "*Visitar el sitio web de Fertilicalc*" redirige al usuario al sitio web oficial de Fertilicalc, el elemento "*Ver vídeo tutorial*" redirige al usuario a un video tutorial de Youtube, y el elemento "*Funciones futuras*" muestra un cuadro de diálogo con una lista de mejoras que pronto se implementarán.
- ***Feedback.*** El elemento "*Dejar feedback*" redirige al usuario a una página de *Google Forms*.

La pantalla *Languages* muestra la lista de idiomas disponibles para la aplicación. Cuando el usuario cambia de idioma, esta pantalla llama a los comandos apropiados para activar el cambio y almacenar la elección realizada.

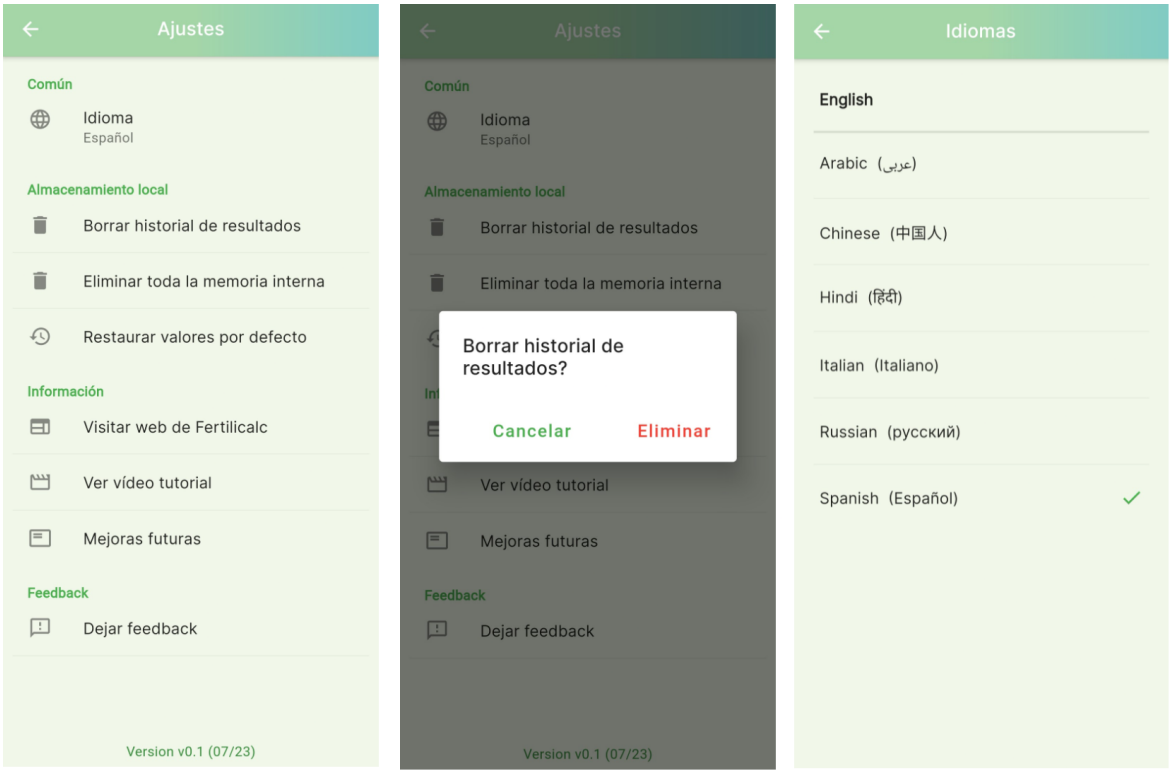


Figura 5-12. Pantalla *Settings* y pantalla *Languages*

5.3 Modelo

Los componentes del módulo *Modelo* encapsulan la mayoría de los datos que se muestran en las pantallas. Incluyen principalmente dos tipos de clases: 'objetos de datos normales' y 'notificadores de cambios'. Las primeras son clases simples que encapsulan objetos (p.ej., un cultivo junto con todas sus propiedades). Las otras almacenan listas de objetos (p.ej., cultivos), junto con datos adicionales, y definen algunas funciones básicas para filtrar y manipular estos datos. Extienden *ChangeNotifier* para poder notificar a los oyentes cuando sus datos han cambiado, a través del mecanismo de *Provider*.

A continuación, se muestra un fragmento del diagrama de clases del módulo, donde las clases *ChangeNotifier* están remarcadas en color:

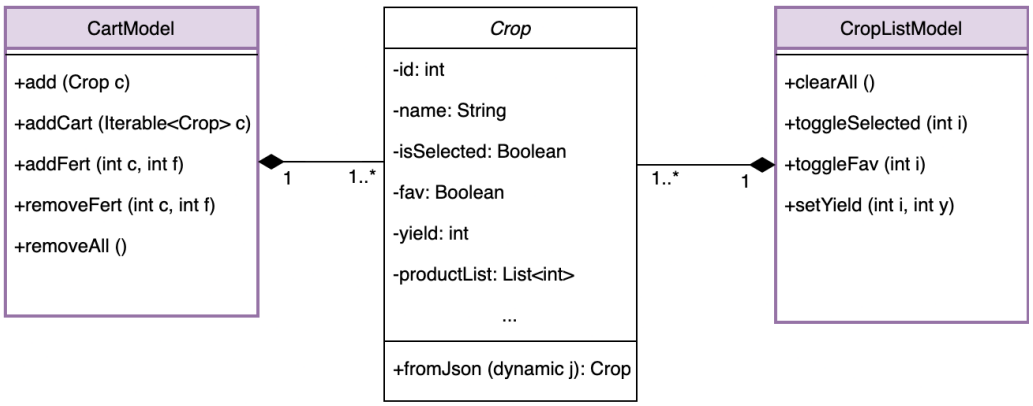


Figura 5-13. Diagrama de clases del modulo Modelo (fragmento)

Para mayor simplicidad, hemos optado por mantener una lógica básica dentro de estas clases. Por ejemplo, algunas de las clases de datos simples tienen métodos que permiten convertir el objeto a JSON o viceversa, y algunas de las clases *ChangeNotifier* tienen métodos que realizan operaciones simples con sus datos.

5.4 Servicios y *Backend*

El módulo Servicios interactúa con cualquier cosa fuera del entorno (*sandbox*) de la aplicación. En nuestro caso, esto implica interactuar con el almacenamiento local del dispositivo y con los servicios de Internet de los que depende la aplicación. Los Comandos llaman a los Servicios para realizar estas interacciones y potencialmente actualizar el Modelo.

El siguiente diagrama ilustra la integración de la arquitectura de nuestra aplicación con el *backend* con el que se comunican los Servicios:

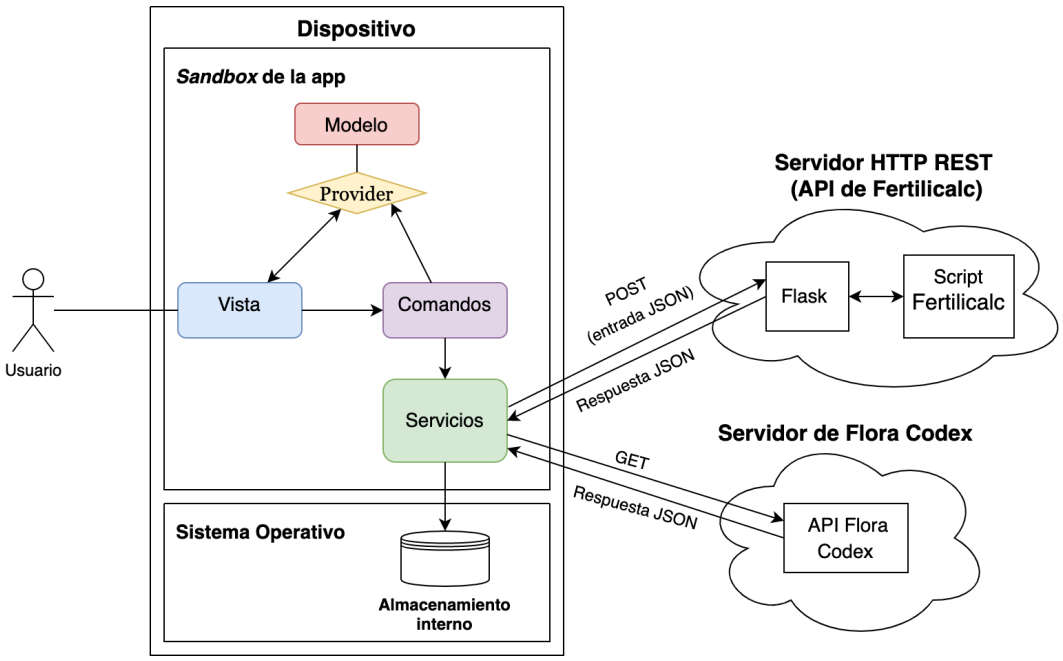


Figura 5-14. Diagrama de sistemas de la aplicación y el *backend*

El servidor de la API REST que aloja el script de Fertilicalc fue configurado por nosotros (su funcionamiento se describe en el Apéndice A de la memoria). La siguiente figura ilustra el proceso de interacción con nuestra API:

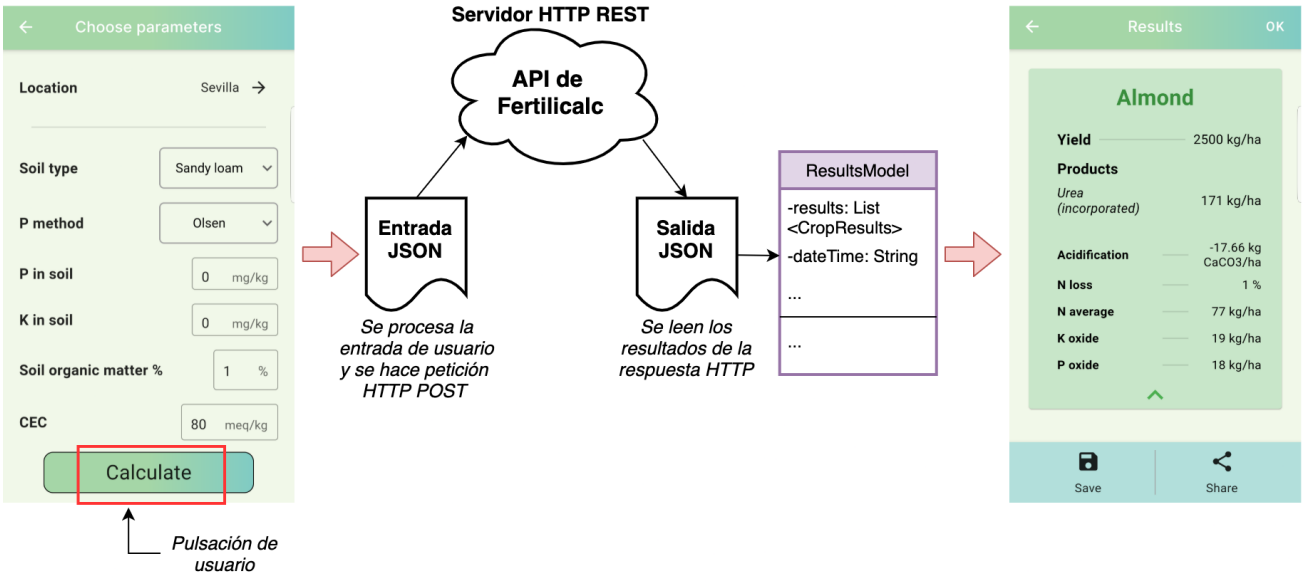


Figura 5-15. Interacción con la API de Fertilicalc

La API Flora Codex es un repositorio de información botánica de fácil acceso que cuenta con una colección exhaustiva de plantas. Decidimos utilizarlo para reforzar la componente educativa de la aplicación. La siguiente figura ilustra la interacción con esta API:

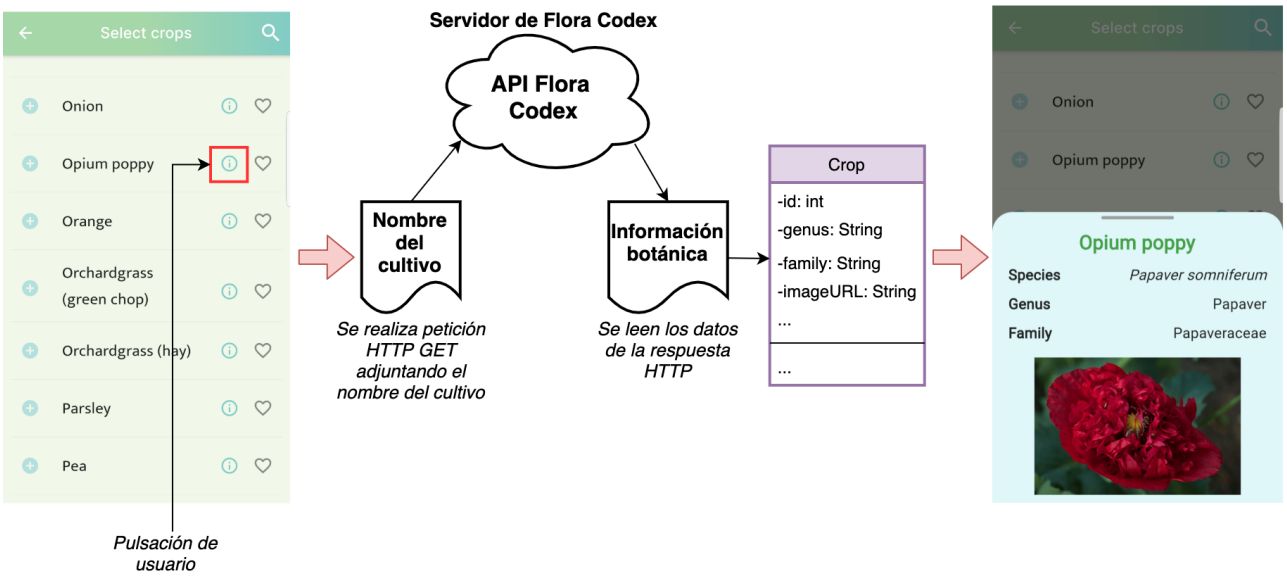


Figura 5-16. Interacción con la API Flora Codex

Respecto a la interacción con el almacenamiento interno, cabe destacar que usamos el *plugin SharedPreferences*, que permite guardar tipos simples de datos tanto en Android como iOS. Hemos escogido esta solución por su versatilidad y facilidad de uso. Aunque sólo nos permite tratar tipos simples (*int*, *double*, *string*, etc.), es posible guardar objetos más complejos si codificamos objetos JSON anidados como *strings*. Así es como logramos, por ejemplo, guardar los resultados en el historial.

5.5 Comandos

Los comandos implementan la lógica que interactúa con diferentes conjuntos de datos, responden a eventos desencadenados por el usuario, y se comunican con servicios externos. Esencialmente, actúan como pegamento entre las diferentes capas, junto con *Provider*, que une los datos directamente entre la Vista y el Modelo.

Un ejemplo es el comando `LoadResults`, que se encarga del cálculo de nuevos resultados y sigue el siguiente flujo de ejecución:

- Reúne la entrada del usuario (lista de cultivos y fertilizantes, y parámetros), leyéndola del *Provider*.
- Llama a una función auxiliar que, a partir de la entrada del usuario, devuelve la cadena JSON adecuada para adjuntar a la petición HTTP.
- Llama al servicio `fetchFromAPI`, pasándole la entrada JSON; después, recibe la respuesta, que debería contener los resultados del cálculo.
- Si la llamada al servicio fue exitosa, el modelo *ResultsModel* se actualiza con la información devuelta. Otra función auxiliar es llamada para convertir la respuesta en JSON al objeto de Dart.

Una vez que los resultados se han cargado en el *Provider* (escribiéndolos en el *ResultsModel*), se redirige al usuario a la pantalla *Results*. Esta pantalla lee los resultados del *ResultsModel* y los muestra.

Este desacoplamiento de datos (Modelo), lógica (Comandos y Servicios) y presentación (Vista) nos permite reutilizar la pantalla *Results* para mostrar los resultados guardados, además de los recién calculados.

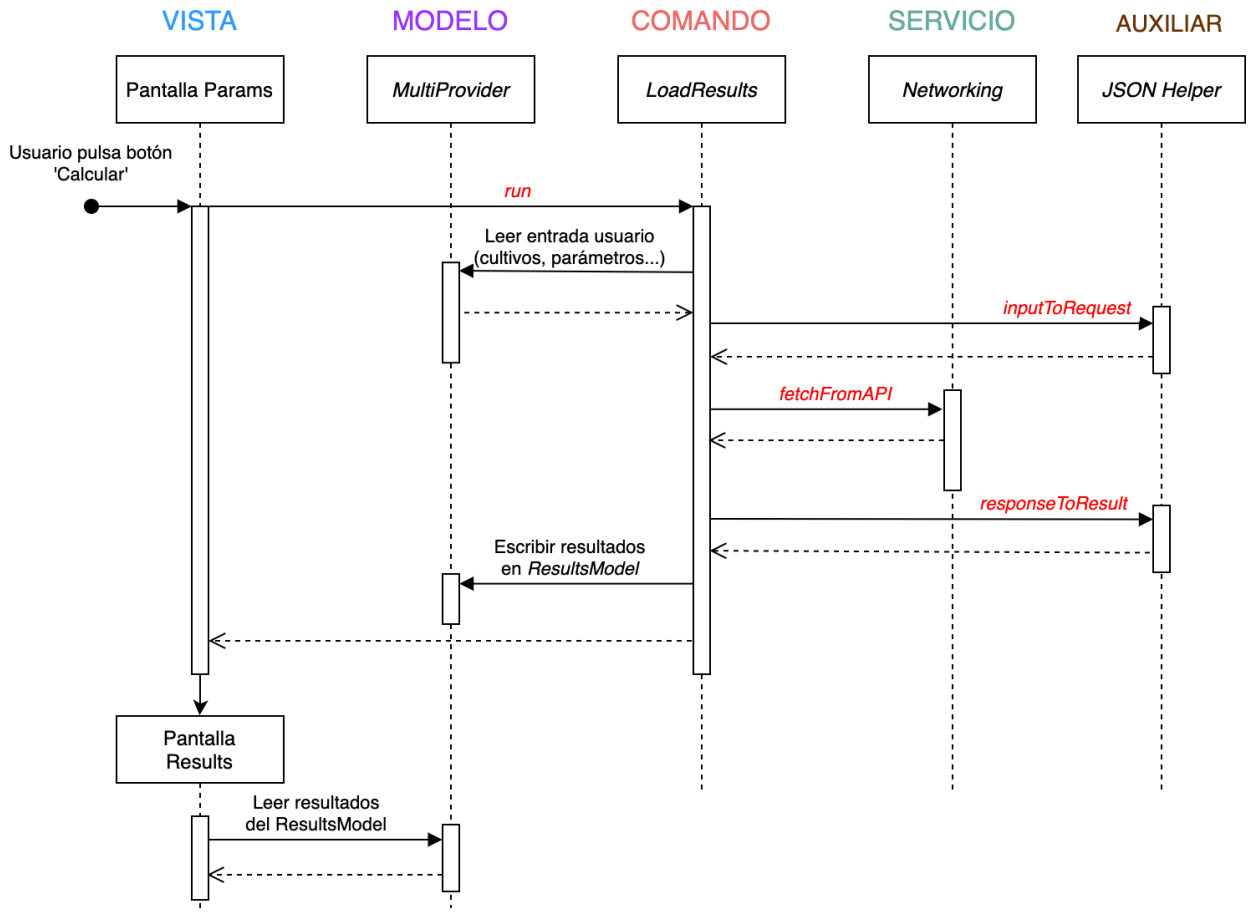


Figura 5-17. Diagrama de secuencia del cálculo de resultados

En la memoria completa se incluyen más detalles sobre los diferentes comandos definidos.

Mediante la implementación e integración de los diferentes módulos, cumplimos con todos los requisitos propuestos en el capítulo 3. En el Apéndice C de la memoria completa incluimos una tabla que describe las clases de cada módulo involucradas en la entrega de cada requisito.

5.6 Internacionalización

La internacionalización consiste en implementar la aplicación de manera que sea posible traducir textos y diseños para cada idioma soportado por la aplicación. Flutter proporciona widgets y clases que ayudan con la internalización, y las propias bibliotecas de Flutter están internacionalizadas.

Hemos incluido 7 idiomas: inglés, español, italiano, ruso, chino, hindi y árabe. En la aplicación *Fertilicalc* de escritorio, se realizaron numerosas traducciones en colaboración con especialistas extranjeros; hemos incorporado estas traducciones a través de archivos JSON locales y las usamos para mostrar textos traducidos en nuestras pantallas. Usamos un archivo JSON para cada idioma soportado.

Las traducciones incorporadas incluyen nombres de cultivos, fertilizantes, parámetros, unidades y diversos términos técnicos. Sin embargo, hay determinados términos específicos de nuestras pantallas de los que no disponemos de traducciones de tanta calidad. Para traducir estos términos adicionales, hemos usado *Google Translate* y la *Colección de terminología de Microsoft*.

Para refinar la calidad de estar traducciones, hemos creado una hoja de cálculo colaborativa de *Google Sheets*, destinada a facilitar la corrección de las traducciones. El objetivo es que sea compartida con usuarios nativos que deseen colaborar en esta tarea.

Para automatizar el proceso de actualización de las traducciones en la aplicación, hemos escrito un *script* de Python que lee las traducciones corregidas de la hoja de cálculo y actualiza los archivos de idiomas JSON de la aplicación.

El mecanismo que usamos para traducir textos en la aplicación se basa en la biblioteca *flutter_localizations* de Flutter. Esta se encarga de adaptar los widgets a los textos traducidos y al diseño correcto (de izquierda a derecha o de derecha a izquierda). Cuando se cambia el idioma del dispositivo, los widgets de la aplicación se adaptan a esta configuración regional (si el idioma está soportado).

Para permitir que un usuario elija un idioma en particular (R-28), hemos implementado métodos personalizados (descritos en la sección 5.5.5) que son llamados en la pantalla *Languages*.

A continuación, mostramos diferentes pantallas de la aplicación en diferentes idiomas. De izquierda a derecha se muestran hindi, italiano, y árabe:

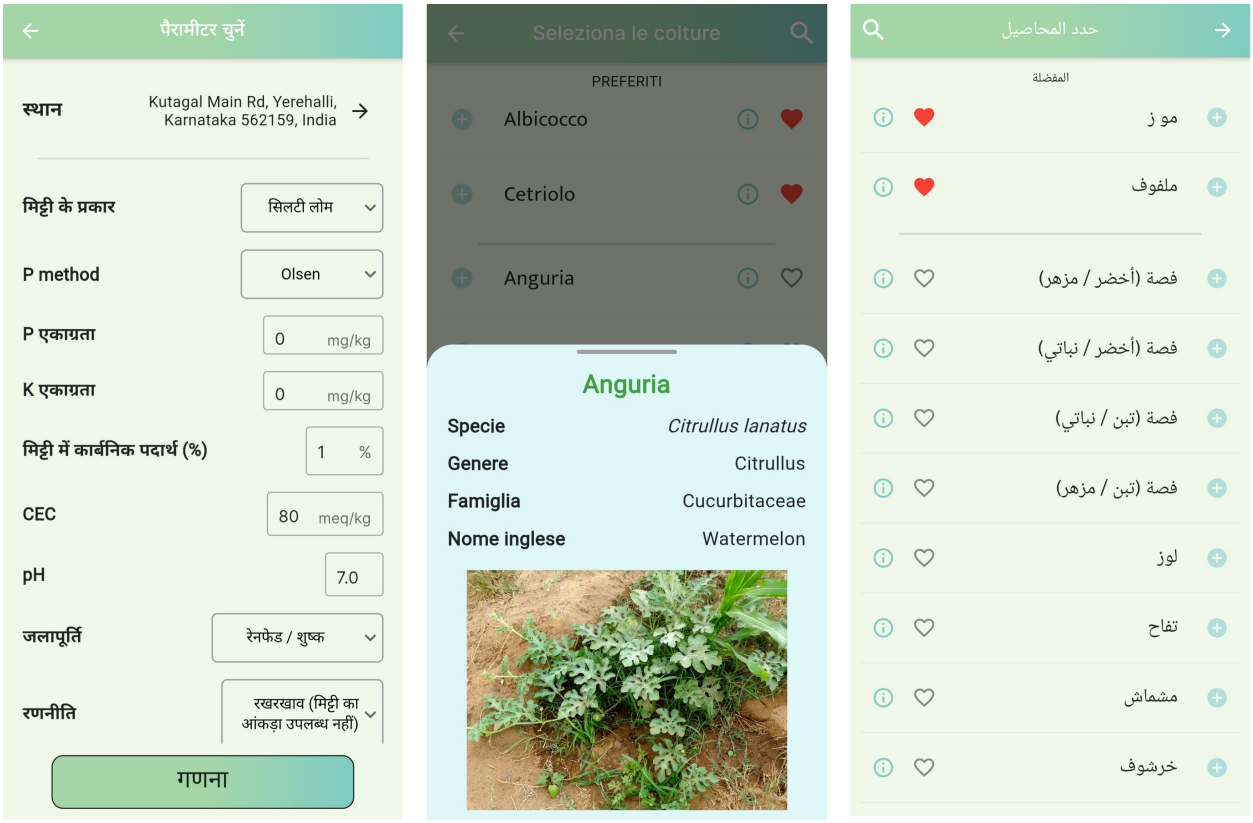


Figura 5-18. Pantallas traducidas

Dado que las bibliotecas de Flutter ya están internacionalizadas, cuando la aplicación se ejecuta en árabe, los widgets de Flutter adoptan automáticamente una disposición de derecha a izquierda.

6 CONCLUSIONES Y LÍNEAS DE MEJORA

Desde agosto de 2021, estoy en proceso de probar la aplicación de Android con '*testers*' seleccionados, a través de la herramienta "*Closed Testing*" que ofrece *Google Play Console*. Hasta ahora, las pruebas han demostrado que nuestra aplicación es estable y se puede ejecutar en varios dispositivos.

Desarrollar este proyecto fue un gran desafío; afortunadamente, la gran cantidad de documentación y recursos disponibles para el desarrollo en Flutter fue increíblemente útil.

Si bien estimo que la curva de aprendizaje hubiera sido similar para Android o iOS, considero que este proyecto valida el uso de Flutter para aplicaciones pequeñas o medianas. Flutter parece tener un futuro prometedor, con una comunidad de desarrolladores apasionados cada vez mayor y un amplio apoyo del equipo de Google.

El proceso de diseño se vio favorecido sustancialmente por la introducción de una arquitectura estructurada. Refactorizar el código para adaptarse a esto no fue particularmente laborioso, principalmente porque durante todo el proyecto mantuvimos nuestro enfoque de gestión de estado.

Se han cumplido todos los requisitos funcionales que se establecieron. El uso de paquetes externos fue de gran ayuda para la implementación de algunos de estos. Considero que también se han cumplido los requisitos no funcionales:

- **Tamaño.** El tamaño del ejecutable actual es de 16,8 MB, por debajo de los 20 MB propuestos.
- **Usabilidad.** La respuesta que hemos recibido hasta ahora ha sido satisfactoria.[3]
- **Internacionalización.** La aplicación se ha adaptado con éxito a los 7 idiomas que propusimos.
- **Compatibilidad.** La aplicación se puede ejecutar en los dispositivos propuestos.

Considero que el objetivo fundamental del proyecto, que era desarrollar la aplicación móvil *Fertilicalc Mobile*, se ha cumplido. Además, el componente educativo de la aplicación se ha reforzado mediante la incorporación de información botánica de la API Flora Codex.

A continuación, exponemos algunas de las áreas en las que consideramos que se debe priorizar el trabajo futuro para mejorar la aplicación:

- **Precios de fertilizantes.** Podría ser de interés mostrar precios aproximados de productos fertilizantes y de las dosis propuestas. Obtener precios a escala global no sería práctico; por otro lado, contactar con distribuidores locales podría ser una opción viable. Agregar publicidad de estos también podría servir como una forma de patrocinar la aplicación. Otra opción más sencilla pero menos precisa sería usar un script para agregar precios obtenidos de *Alibaba*.
- **Integración con la nube.** La adición de una pantalla de inicio de sesión y la integración con el almacenamiento en la nube permitiría a los usuarios acceder a sus datos en cualquier dispositivo que elijan. Sería de especial interés integrar la aplicación con los servicios de *backend* de *Firebase*, debido al amplio soporte y documentación ya disponible.
- **Mantenimiento.** Flutter y Dart son tecnologías relativamente jóvenes. Desde que comenzamos el desarrollo de la aplicación, ha habido actualizaciones significativas en el *framework* y en algunos de los paquetes externos que hemos incorporado. Es imperativo a corto o medio plazo atender a estas actualizaciones. Una de las principales tareas de este mantenimiento es actualizar el código a Flutter 2.
- **Usabilidad.** Una mejora importante sería incluir mensajes informativos en diferentes puntos de la aplicación para explicar al usuario algunos de los términos técnicos y la funcionalidad asociada. Otra opción sería incluir un breve *walk-through* al inicio la aplicación que muestre un recorrido por la aplicación. También proponemos medir la usabilidad más a fondo, por ejemplo, a través de una encuesta.

[3] Para reforzar este aspecto, hemos elaborado tutoriales en inglés (https://youtu.be/KBc7tqCvHCI) y español (https://youtu.be/zTfh5sAfrhY)